622.设计循环队列

实现支持如下操作：

- MyCircularQueue(k): 构造器，设置队列长度为 k 。
- Front: 从队首获取元素。如果队列为空，返回 -1 。
- Rear: 获取队尾元素。如果队列为空，返回 -1 。
- enQueue(value):向循环队列插入一个元素。如果成功插入则返回真。
- deQueue(): 从循环队列中删除一个元素。如果成功删除则返回真。
- isEmpty(): 检查循环队列是否为空。
- isFull(): 检查循环队列是否已满。

```
class MyCircularQueue {
private:
    // store elements
    vector<int> data;
    // a pointer to indicate the start and end position
    int p_start;
    int p_end;
    int size;
    int capacity;
public:
    /** Initialize your data structure here. Set the size of the queue to be k. */
    MyCircularQueue(int k) {
        data.reserve(k);
        p_start = 0;
        p_end = 0;
        size = 0;
        capacity = k;
    }

    /** Insert an element into the circular queue. Return true if the operation is
successful. */
    bool enQueue(int value) {
        if(isFull())
            return false;
        data[p_end] = value;
        p_end = (p_end + 1) % capacity;
        size++;
        return true;
    }

    /** Delete an element from the circular queue. Return true if the operation is
successful. */
    bool deQueue() {
        if(isEmpty())
            return false;
        p_start = (p_start + 1) % capacity;
        size--;
```

```
        return true;
    }

    /** Get the front item from the queue. */
    int Front() {
        if(isEmpty())
            return -1;
        return data[p_start];
    }

    /** Get the last item from the queue. */
    int Rear() {
        if(isEmpty())
            return -1;
        return data[(p_end+capacity-1)%capacity];
    }

    /** Checks whether the circular queue is empty or not. */
    bool isEmpty() {
        return size==0;
    }

    /** Checks whether the circular queue is full or not. */
    bool isFull() {
        return size == capacity;
    }
};
```

C++ STL 常见操作

```
#include <iostream>

int main() {
    // 1. Initialize a queue.
    queue<int> q;
    // 2. Push new element.
    q.push(5);
    q.push(13);
    q.push(8);
    q.push(6);
    // 3. Check if queue is empty.
    if (q.empty()) {
        cout << "Queue is empty!" << endl;
        return 0;
    }
    // 4. Pop an element.
    q.pop();
    // 5. Get the first element.
    cout << "The first element is: " << q.front() << endl;
    // 6. Get the last element.
    cout << "The last element is: " << q.back() << endl;
```

```
    // 7. Get the size of the queue.
    cout << "The size is: " << q.size() << endl;
}
```

使用queue实现广度优先算法伪代码模板

使用used(可用哈希表实现)确保每个节点只被访问一次，避免无限循环

```
/**
 * Return the length of the shortest path between root and target node.
 */
int BFS(Node root, Node target) {
    Queue<Node> queue;  // store all nodes which are waiting to be processed
    Set<Node> used;     // store all the used nodes
    int step = 0;       // number of steps neeeded from root to current node
    // initialize
    add root to queue;
    add root to used;
    // BFS
    while (queue is not empty) {
        step = step + 1;
        // iterate the nodes which are already in the queue
        int size = queue.size();
        for (int i = 0; i < size; ++i) {
            Node cur = the first node in queue;
            return step if cur is target;
            for (Node next : the neighbors of cur) {
                if (next is not in used) {
                    add next to queue;
                    add next to used;
                }
            }
            remove the first node from queue;
        }
    }
    return -1;          // there is no path from root to target
}
```

有两种情况你不需要使用哈希集：

- 你完全确定没有循环，例如，在树遍历中
- 你确实希望多次将结点添加到队列中

200.岛屿的数量（中等）

1. 题目描述

给定一个由 '1'（陆地）和'0'（水）组成的的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

示例 1:

```
输入:
11110
11010
11000
00000

输出: 1
```

示例 2:

```
输入:
11000
11000
00100
00011

输出: 3
```

2. 简单实现 广度优先遍历，每次将每个点上下左右相邻的非0点放入队列，即连接的一片岛屿，每次队列空了就是一边岛屿遍历完了

```
class Solution {
private:
public:
    int numIslands(vector<vector<char>>& grid) {
        int m = grid.size();
        if(m == 0)
            return 0;
        int n = grid[0].size();
        queue<vector<int>> q;
        int i,j;
        int count = 0;
        for(i = 0; i < m; i++){
            for(j = 0; j < n; j++){
                if(grid[i][j] == '1'){//发现新岛屿
                    q.push({i,j});
                    grid[i][j] = '0';//访问过该节点
                    count++;
                    while(!q.empty()){
                        vector<int> cur = q.front();
                        q.pop();
                        int x = cur[0], y = cur[1];
                        if(x>0 && grid[x-1][y] == '1'){
                            grid[x-1][y] = '0';
                            q.push({x-1,y});
                        }
                        if(y>0 && grid[x][y-1] == '1'){
                            grid[x][y-1] = '0';
                            q.push({x,y-1});
                        }
                        if(x<m-1 && grid[x+1][y] == '1'){
                            grid[x+1][y] = '0';
```

```
                            q.push({x+1,y});
                        }
                        if(y<n-1 && grid[x][y+1] == '1'){
                            grid[x][y+1] = '0';
                            q.push({x,y+1});
                        }
                    }
                }
            }
        }
        return count;
    }
};
```

752.打开转盘锁（中等）

1. 题目描述

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' 。每个拨轮可以自由旋转：例如把 '9' 变为 '0'，'0' 变为 '9' 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000' ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

示例 1:

```
输入: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出: 6
解释:
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
因为当拨动到 "0102" 时这个锁就会被锁定。
```

示例 2:

```
输入: deadends = ["8888"], target = "0009"
输出: 1
解释:
把最后一位反向旋转一次即可 "0000" -> "0009"。
```

示例 3:

```
输入: deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target =
"8888"
输出: -1
解释:
无法旋转到目标数字且不被锁定。
```

示例 4:

```
输入: deadends = ["0000"], target = "8888"
输出: -1
```

提示:

- 死亡列表 deadends 的长度范围为 [1, 500]。
- 目标数字 target 不会在 deadends 之中。
- 每个 deadends 和 target 中的字符串的数字会在 10,000 个可能的情况 '0000' 到 '9999' 中产生。

2. 简单实现

将死亡数看做已被访问的str即可，每次可前后拨动四位中的某一位

```
class Solution {
public:
    int openLock(vector<string>& deadends, string target) {
        //初始化
        bool visited[10000] = {false};
        for(int i = 0; i < deadends.size(); i++){//将死亡数设置成已访问
            string str = deadends[i];
            if(str == "0000")
                return -1;
            visited[stoi(str)] = true;
        }
        int count = -1;
        string str = "0000";
        queue<string> q;
        q.push(str);
        visited[stoi(str)] = true;
        //遍历
        while(!q.empty()){
            count++;//层次+1，即拨动次数加一
            int size = q.size();
            for(int i = 0; i < size; i++){
                string cur = q.front();
                q.pop();
                if(cur == target)
                    return count;//解锁成功
                else{
                    for(int idx = 0; idx < 4; idx++){//四个密码位
                        string temp = cur;
                        char x = temp[idx] + 1;//向后拨
                        if(x > '9') x = '0';
                        temp[idx] = x;
                        if(!visited[stoi(temp)]) q.push(temp);
                        visited[stoi(temp)] = true;

                        temp = cur;
                        x = temp[idx] - 1;//向前拨
                        if(x < '0') x = '9';
                        temp[idx] = x;
```

```
                        if(!visited[stoi(temp)]) q.push(temp);
                        visited[stoi(temp)] = true;
                    }
                }
            }
        }
        return -1;
    }
};
```

Tips:要在字符串进队列时就设置visited，不能在cur处设置，否则会有可能在else里将相同元素放入队列

279.完全平方数（中等）

1. 题目描述

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

示例 1:

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
```

示例 2:

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9.
```

2. 简单实现

队列q内存储n每一轮减去一个完全平方数之后剩余的部分

```
class Solution {
public:
    int numSquares(int n) {
        queue<int> q;
        q.push(n);
        int count = 0;
        while(1){
            count++;
            int size = q.size();
            for(int i = 0; i < size; i ++){
                int cur = q.front();
                q.pop();
                int sqrts = int(sqrt(cur));
                if(sqrts*sqrts == cur)//减完了
                    return count;
                for(int k = sqrts; k > 0; k--){
```

```
                    q.push(cur-k*k);
                }
            }
        }
        return -1;
    }
};
```

3. 动态规划

- 首先初始化长度为n+1的数组dp，每个位置都为0，如果n为0，则结果为0
- 对数组进行遍历，下标为i，每次都将当前数字先更新为最大的结果，即dp[i]=i
- 动态转移方程为: dp[i] = MIN(dp[i], dp[i - j * j] + 1)，i表示当前数字，j*j表示平方数
- 时间复杂度：O(n*sqrt(n))，sqrt为平方根

```
class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1]; // 默认初始化值都为0
        for (int i = 1; i <= n; i++) {
            dp[i] = i; // 最坏的情况就是每次+1
            for (int j = 1; i - j * j >= 0; j++) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1); // 动态转移方程
            }
        }
        return dp[n];
    }
}
```

4. 最优实现（没看懂的数学方法）

```
class Solution {
public:
    int numSquares(int n) {
        while (!(n & 3)) {
            n = n >> 2;
        }
        if (n % 8 == 7)
            return 4;
        int a = 0;
        while (a*a < n) {
            int b = sqrt(n - a*a);
            if (a*a + b*b == n) {
                return ((int)(a != 0)) + ((int)(b != 0));
            }
            a++;
        }
        return 3;
    }
};
```

C++ STL stack 常见caozuo

```
#include <iostream>
int main() {
    // 1. Initialize a stack.
    stack<int> s;
    // 2. Push new element.
    s.push(5);
    s.push(13);
    // 3. Check if stack is empty.
    if (s.empty()) {
        cout << "Stack is empty!" << endl;
        return 0;
    }
    // 4. Pop an element.
    s.pop();
    // 5. Get the top element.
    cout << "The top element is: " << s.top() << endl;
    // 6. Get the size of the stack.
    cout << "The size is: " << s.size() << endl;
}
```

155.最小栈（简单）

1. 题目描述

设计一个支持 push，pop，top操作，并能在常数时间内检索到最小元素的栈。

- push(x) -- 将元素 x 推入栈中。
- pop() -- 删除栈顶的元素。
- top() -- 获取栈顶元素。
- getMin() -- 检索栈中的最小元素。 示例:

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

2. 简单实现

在使用正常栈的同时维护一个最小栈，每次入栈时元素若小于最小栈栈顶元素则压入最小栈，每次出栈时元素若等于最小栈栈顶元素则最小栈出栈

```
class MinStack {
public:
    /** initialize your data structure here. */
    MinStack() {
```

```
    }

    void push(int x) {
        if( minStack.empty() || x <= minStack.top())
            minStack.push(x);
        basicStack.push(x);
    }

    void pop() {
        if (basicStack.top() == minStack.top())
            minStack.pop();
        basicStack.pop();
    }

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

    int getMin() {
        return minStack.top();
    }
    stack<int> basicStack;
    stack<int> minStack;
};
```

20.有效的括号（简单）

1. 题目描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。
- 注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入: "{[]}"
输出: true
```

2. 简单实现

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> judge;
        int len = s.length();
        if(len%2==1) return false;//奇数一定不匹配
        for(int i = 0; i<len; i++)
        {
            char cur = s[i];
            if(cur=='(' || cur=='[' || cur=='{')
                judge.push(cur);
            else if(cur==')')
            {
                 if(!judge.empty()&&judge.top()=='(')
                    judge.pop();
                 else
                    return false;
            }
            else if(cur==']')
            {
                 if(!judge.empty()&&judge.top()=='[')
                    judge.pop();
                 else
                    return false;
            }
             else if(cur=='}')
            {
                 if(!judge.empty()&&judge.top()=='{')
                    judge.pop();
                 else
                    return false;
            }
        }
        if(judge.empty())
            return true;
        else
            return false;
    }
};
```

739.每日温度（中等）

1. 题目描述

根据每日 气温 列表，请重新生成一个列表，对应位置的输入是你需要再等待多久温度才会升高超过该日的天数。如果之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

提示： 气温 列表长度的范围是 [1, 30000]。每个气温的值的均为华氏度，都是在 [30, 100] 范围内的整数。

2. 简单实现

- 使用O(n^2)的暴力遍历法会超出时间限制，需要寻找O(n)的算法
- 采用最小栈即可解决

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& T) {
        vector<int> ans(T.size(), 0);
        int len = T.size();
        if(len == 1)
            return vector<int>(1,0);

        stack<vector<int>> minstack;//最小栈:<val, idx>
        for(int i = 0; i < len; i++){
            if(minstack.empty() || T[i] <= minstack.top()[0]){
                minstack.push({T[i],i});
            }
            else{
                while(!minstack.empty() && T[i] > minstack.top()[0]){
                    vector<int> cur = minstack.top();
                    ans[cur[1]] = i-cur[1];
                    minstack.pop();
                }
                if(minstack.empty() || T[i] <= minstack.top()[0])
                    minstack.push({T[i],i});
            }
        }
        return ans;
    }
};
```

3. 最优性能

直接用最小栈存T中的idx，不用存val啊。。。

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& T) {
        stack<int> stk;
        stk.push(0);
        vector<int> ret(T.size(), 0);
```

```
        for(int i = 1; i < T.size(); ++i){
            while(!stk.empty() && T[i] > T[stk.top()]){
                ret[stk.top()] = i - stk.top();
                stk.pop();
            }
            stk.push(i);
        }
        return ret;
    }
};
```

150.逆波兰表达式求值（中等）

1. 题目描述

根据逆波兰表示法，求表达式的值。有效的运算符包括 +, -, *, /，每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
输入: ["2", "1", "+", "3", "*"]
输出: 9
解释: ((2 + 1) * 3) = 9
```

示例 2：

```
输入: ["4", "13", "5", "/", "+"]
输出: 6
解释: (4 + (13 / 5)) = 6
```

示例 3：

```
输入: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
输出: 22
解释:
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

2. 简单实现

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> s;
        for(int i = 0; i < tokens.size(); i++){
            string temp = tokens[i];
            if(temp == "+"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a+b);
            }
            else if(temp == "-"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a-b);
            }
            else if(temp == "*"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a*b);
            }
            else if(temp == "/"){
                int b = s.top();
                s.pop();
                int a = s.top();
                s.pop();
                s.push(a/b);
            }
            else
                s.push(stoi(temp));
        }
        return s.top();
    }
};
```

使用stack实现深度优先搜索伪代码模板

Tip: DFS最先访问到的不一定是最短路径

1. 递归

```
/*
 * Return true if there is a path from cur to target.
 */
boolean DFS(Node cur, Node target, Set<Node> visited) {
    return true if cur is target;
    for (next : each neighbor of cur) {
        if (next is not in visited) {
            add next to visted;
            return true if DFS(next, target, visited) == true;
        }
    }
    return false;
}
```

2. 显示栈

```
/*
 * Return true if there is a path from cur to target.
 */
boolean DFS(int root, int target) {
    Set<Node> visited;
    Stack<Node> s;
    add root to s;
    while (s is not empty) {
        Node cur = the top element in s;
        return true if cur is target;
        for (Node next : the neighbors of cur) {
            if (next is not in visited) {
                add next to s;
                add next to visited;
            }
        }
        remove cur from s;
    }
    return false;
}
```

岛屿数量

是前面queue中的题，用递归DFS实现

```
class Solution {
public:
    void DFS(vector<vector<char>>& grid, int x, int y){
        int m = grid.size();
        int n = grid[0].size();
        grid[x][y] = '0';
        if(x>0 && grid[x-1][y] == '1'){
           DFS(grid, x-1, y);
        }
```

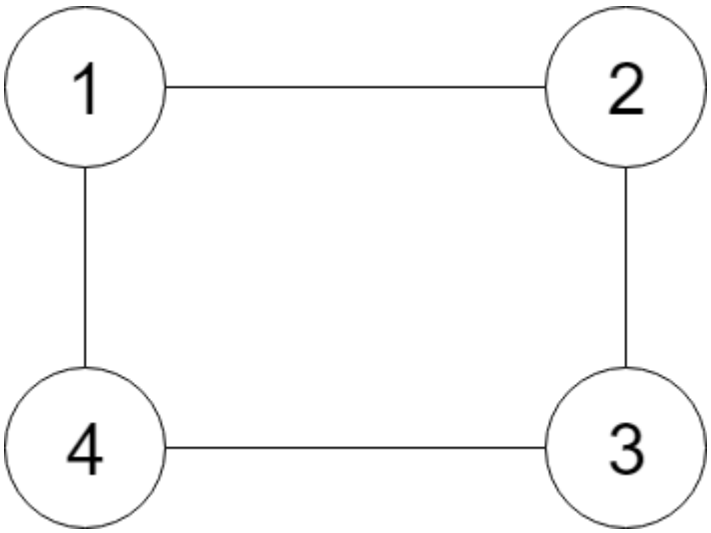
```
        if(y>0 && grid[x][y-1] == '1'){
            DFS(grid, x, y-1);
        }
        if(x<m-1 && grid[x+1][y] == '1'){
            DFS(grid, x+1, y);
        }
        if(y<n-1 && grid[x][y+1] == '1'){
            DFS(grid, x, y+1);
        }
}
    int numIslands(vector<vector<char>>& grid) {
        int m = grid.size();
        if(m == 0)
            return 0;
        int n = grid[0].size();

        int i,j;
        int count = 0;
        for(i = 0; i < m; i++){
            for(j = 0; j < n; j++){
                if(grid[i][j] == '1'){//发现新岛屿
                    count++;
                    DFS(grid, i, j);
                }
            }
        }
        return count;
    }
};
```

133.克隆图（中等）

1. 题目描述

给定无向连通图中一个节点的引用，返回该图的深拷贝（克隆）。图中的每个节点都包含它的值 val（Int） 和其邻居的列表（list[Node]）。

示例：




```
输入：
{"$id":"1","neighbors":[{"$id":"2","neighbors":[{"$ref":"1"},{"$id":"3","neighbors":
[{"$ref":"2"},{"$id":"4","neighbors":[{"$ref":"3"},
{"$ref":"1"}],"val":4}],"val":3}],"val":2},{"$ref":"4"}],"val":1}

解释：
节点 1 的值是 1，它有两个邻居：节点 2 和 4 。
节点 2 的值是 2，它有两个邻居：节点 1 和 3 。
节点 3 的值是 3，它有两个邻居：节点 2 和 4 。
节点 4 的值是 4，它有两个邻居：节点 1 和 3 。
```

提示：

- 节点数介于 1 到 100 之间。
- 无向图是一个简单图，这意味着图中没有重复的边，也没有自环。
- 由于图是无向的，如果节点 p 是节点 q 的邻居，那么节点 q 也必须是节点 p 的邻居。
- 必须将给定节点的拷贝作为对克隆图的引用返回。

2. 简单实现

难点在于不能直接new Node(node->val, node->neighbors)，这样只深拷贝了val，没有深拷贝neighbors，因此必须有所记录

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> neighbors;

    Node() {}

    Node(int _val, vector<Node*> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
};
```

```
*/
class Solution {
public:
    map<Node*,Node*> mp;
    Node* cloneGraph(Node* node) {
        if(!node)   return NULL;
        if(mp.count(node))  return mp[node];
        Node* c = new Node(node -> val, {});
        mp[node] = c;
        for(int i = 0; i < node -> neighbors.size(); ++ i){
            if(node -> neighbors[i])    c -> neighbors.push_back(cloneGraph(node ->
neighbors[i]));
        }
        return c;
    }
};
```

494.目标和（中等）

1. 题目描述

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例 1:

```
输入: nums: [1, 1, 1, 1, 1], S: 3
输出: 5

解释:
-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3
一共有5种方法让最终目标和为3。
```

注意:

- 数组非空，且长度不会超过20。
- 初始的数组的和不会超过1000。
- 保证返回的最终结果能被32位整数存下。

2. 简单实现

企图用全局变量的逻辑控制会非常复杂，利用递归时，要把思想变简单，多用传参

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int S) {
```

```
        int res = 0, sum = 0;
        dfs(nums, S, 0, sum, res);
        return res;
    }

    void dfs(vector<int>& nums, int S, int i, int sum, int& res) {
        if (i == nums.size()) {
            if (sum == S) {
                ++res;
            }
            return;
        }
        dfs(nums, S, i + 1, sum + nums[i], res);
        dfs(nums, S, i + 1, sum - nums[i], res);
    }
};
```

94.二叉树的中序遍历（中等）

1. 题目描述

给定一个二叉树，返回它的中序 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,3,2]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

2. 简单实现

一个要点在于需要有一个flag标识栈顶元素的左子树是否遍历完

```
class Solution {
public:
    vector<int>  inorderTraversal(TreeNode* root) {
        vector<int> ans;
        if(!root)
            return ans;
        stack<TreeNode*> s;
        s.push(root);
        bool left_done = false;
        while(!s.empty()){
            if(!left_done){
                while(s.top()->left)
```

```
                s.push(s.top()->left);
            }
            TreeNode* temp = s.top();
            s.pop();
            ans.push_back(temp->val);
            if(temp->right){
                s.push(temp->right);
                left_done = false;//遍历右子树，右子入栈了
            }
            else
                left_done = true;//左子到达叶子节点了，此时栈顶元素的左子树遍历完毕
        }
        return ans;
    }
};
```

144.二叉树的前序遍历（中等）

1. 题目描述

给定一个二叉树，返回它的前序遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,2,3]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

2. 简单实现

同中序遍历，需要标识左子树的情况

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> ans;
        if(!root)
            return ans;
        stack<TreeNode*> s;
        s.push(root);
        bool left_done = false;
        while(!s.empty()){
            if(!left_done){
                ans.push_back(s.top()->val);
                while(s.top()->left){
```

```
                    s.push(s.top()->left);
                    ans.push_back(s.top()->val);
                }
            }
            TreeNode* temp = s.top();
            s.pop();
            if(temp->right){
                s.push(temp->right);
                left_done = false;
            }
            else{
                left_done = true;
            }
        }
        return ans;
    }
};
```

145.二叉树的后序遍历（困难）

1. 题目描述

给定一个二叉树，返回它的后序遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [3,2,1]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

2. 简单实现

困难在于要同时记录左右子的完成情况，逻辑复杂了（始终记住left_done和right_done表示当前栈顶元素的左右子树完成情况）

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        if(!root)
            return ans;
        stack<TreeNode*> s;
        s.push(root);
        bool left_done = false;
        bool right_done = false;
```

```
        TreeNode* pre;
        while(!s.empty()){
            if(!left_done){
                while(s.top()->left)
                    s.push(s.top()->left);
            }
            if(!right_done && s.top()->right){
                s.push(s.top()->right);
                left_done = false;
                right_done = false;
            }
            else{
                TreeNode* temp = s.top();
                s.pop();
                ans.push_back(temp->val);
                //后序遍历，栈顶元素必然是当前出栈元素的父节点
                if(!s.empty() && s.top()->left == temp){
                    left_done = true;
                    right_done = false;
                }
                if(!s.empty() && s.top()->right == temp){
                    left_done = true;
                    right_done = true;
                }
            }
        }
        return ans;
    }
};
```

练习题

394.字符串解码（中等）

1. 题目描述

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

示例:

```
s = "3[a]2[bc]", 返回 "aaabcbc".
s = "3[a2[c]]", 返回 "accaccacc".
s = "2[abc]3[cd]ef", 返回 "abcabccdcdcdef".
```

2. 简单实现

利用栈一层层解码[]，对码值利用规则进行语义分割（'['，']'以及数字处）并存如栈cache中，当前分割的语义段保存在temp中

潜规则：

- '['前一定是数字
- ']'前一定是字符串
- 数字前可能有字符串，字符串前不可能有数字
- 栈空时，说明当前没有待处理的[]，即目前已消除[]的层叠
- 遍历到字符串时当前栈空，说明当前字符串不在[]中

```
class Solution {
public:
    string decodeString(string s) {
        string ans = "";//答案
        stack<string> cache;//栈
        string temp = "";//统计当前语义相连的数字或字符串
        for(int i = 0; i < s.size(); i++){
            if(s[i] >= '0' && s[i] <= '9'){//是数字
                if(temp != "" && (temp[0] < '0' || temp[0] > '9')){//前面有字符串
                    cache.push(temp);
                    temp = "";
                }
                //统计数字部分
                temp += s[i];
            }
            else if(s[i] != '[' && s[i] != ']'){//是字符串，根据规则，前面不可能有数字
                if(!cache.empty())//在[]中的字符串，需要统计
                    temp += s[i];
                else//在[]外，直接输出至ans
                    ans += s[i];
            }
            else if(s[i] == '['){
                cache.push(temp);
                temp = "";
            }
            else{//是']'
                string cur = "";//存储当前[]的解码字符串
                int times = stoi(cache.top());//栈顶保存temp的重复次数
                cache.pop();
                for(int cnt = 0; cnt < times; cnt++)
                    cur += temp;
                if(cache.empty()){//当前所有[]全部解码，与后续编码无关，更新ans和temp
                    ans += cur;
                    temp = "";
                }
                else{//外面还有至少一层[]
                    string top = cache.top();
                    if(top[0] < '0' || top[0] > '9'){//与前面的字符串合并
                        cache.pop();
                        temp = top + cur;//更新temp
                    }
                    else
```

```
                        temp = cur;//更新temp
                    }
                }
            }
            return ans;
        }
    };
```

733.图像渲染（简单）

1. 题目描述

有一幅以二维整数数组表示的图画，每一个整数表示该图画的像素值大小，数值在 0 到 65535 之间。给你一个坐标 (sr, sc) 表示图像渲染开始的像素值（行 ，列）和一个新的颜色值 newColor，让你重新上色这幅图像。

为了完成上色工作，从初始坐标开始，记录初始坐标的上下左右四个方向上像素值与初始坐标相同的相连像素点，接着再记录这四个方向上符合条件的像素点与他们对应四个方向上像素值与初始坐标相同的相连像素点，……，重复该过程。将所有有记录的像素点的颜色值改为新的颜色值。

最后返回经过上色渲染后的图像。

示例 1:

```
输入:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
输出: [[2,2,2],[2,2,0],[2,0,1]]
解析:
在图像的正中间，(坐标(sr,sc)=(1,1)),
在路径上所有符合条件的像素点的颜色都被更改成2。
注意，右下角的像素没有更改为2，
因为它不是在上下左右四个方向上与初始点相连的像素点。
```

注意:

- image 和 image[0] 的长度在范围 [1, 50] 内。
- 给出的初始点将满足 0 <= sr < image.length 和 0 <= sc < image[0].length。
- image[i][j] 和 newColor 表示的颜色值在范围 [0, 65535]内。

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int
newColor) {
        int raw = image[sr][sc];
        if(raw == newColor)
            return image;
        image[sr][sc] = newColor;
        queue<vector<int>> q;
        q.push({sr,sc});
        while(!q.empty()){
```

```
            int x = q.front()[0];
            int y = q.front()[1];
            q.pop();
            if(x-1 >= 0 && image[x-1][y] == raw){
                image[x-1][y] = newColor;
                q.push({x-1, y});
            }
            if(y-1 >= 0 && image[x][y-1] == raw){
                image[x][y-1] = newColor;
                q.push({x, y-1});
            }
            if(x+1 < image.size() && image[x+1][y] == raw){
                image[x+1][y] = newColor;
                q.push({x+1, y});
            }
            if(y+1 < image[0].size() && image[x][y+1] == raw){
                image[x][y+1] = newColor;
                q.push({x, y+1});
            }
        }
        return image;
    }
};
```

542.01矩阵（中等）

1. 题目描述

给定一个由 0 和 1 组成的矩阵，找出每个元素到最近的 0 的距离。两个相邻元素间的距离为 1 。

示例 1:

```
输入:
0 0 0
0 1 0
0 0 0

输出:
0 0 0
0 1 0
0 0 0
```

示例 2:

```
输入:
0 0 0
0 1 0
1 1 1

输出:
0 0 0
0 1 0
1 2 1
```

注意:

- 给定矩阵的元素个数不超过 10000。
- 给定矩阵中至少有一个元素是 0。
- 矩阵中的元素只在四个方向上相邻: 上、下、左、右。

2. 简单实现

实际上就是多起点同步的BFS，初始化队列将所有为值0的坐标都加入队列，每次BFS扩散一步

```
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
        vector<vector<int>> ans = matrix;
        int m = matrix.size();
        int n = matrix[0].size();
        vector<vector<bool>> finded = vector<vector<bool>>(m, vector<bool>(n, false));
        queue<vector<int>> q;
        int cnt = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(matrix[i][j] == 0){
                    finded[i][j] = true;
                    q.push({i,j});
                }
        while(!q.empty()){
            cnt++;
            int size = q.size();
            for(int i = 0; i < size; i++){
                int x = q.front()[0];
                int y = q.front()[1];
                q.pop();
                if(x-1 >= 0 && !finded[x-1][y]){
                    ans[x-1][y] = cnt;
                    finded[x-1][y] = true;
                    q.push({x-1, y});
                }
                if(y-1 >= 0 && !finded[x][y-1]){
                    ans[x][y-1] = cnt;
                    finded[x][y-1] = true;
                    q.push({x, y-1});
                }
                if(x+1 < m && !finded[x+1][y]){
```

```
                    ans[x+1][y] = cnt;
                    finded[x+1][y] = true;
                    q.push({x+1, y});
                }
                if(y+1 < n && !finded[x][y+1]){
                    ans[x][y+1] = cnt;
                    finded[x][y+1] = true;
                    q.push({x, y+1});
                }
            }
        }
        return ans;
    }
};
```

3. 最优性能代码

整体思路与我相同，大概是几个trick导致速度比我快：

- 用pair<int,int>记录坐标值(比vector快太多!!!)
- 用INT_MAX记录是否访问过，省略finded变量
- 可以利用ans[x][y]已知的情况，省略cnt

```
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
        if ( matrix.size()==0 || matrix[0].size()==0 ) return {};
        int n = matrix.size(), m=matrix[0].size();
        int dir[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};
        queue<pair<int,int>> q;
        vector<vector<int>> res(n, vector<int>(m, 0));
        for ( int i=0; i<n; ++i )
            for ( int j=0; j<m; ++j )
                if ( matrix[i][j]==0 )
                    q.push(make_pair(i,j));
                else
                    res[i][j] = INT_MAX;
        while( !q.empty() ){
            pair<int,int> cor = q.front();
            q.pop();
            for ( int k=0; k<4; ++k ){
                int y = cor.first + dir[k][0];
                int x = cor.second + dir[k][1];
                if ( y<0 || x<0 || y>=n || x>=m || res[y][x]<=res[cor.first]
[cor.second] )
                    continue;
                res[y][x] = res[cor.first][cor.second] + 1;
                q.push(make_pair(y,x));
            }
        }
        return res;
    }
};
```

841.钥匙和房间（中等）

1. 题目描述

有 N 个房间，开始时你位于 0 号房间。每个房间有不同的号码：0，1，2，...，N-1，并且房间里可能有一些钥匙能使你进入下一个房间。

在形式上，对于每个房间 i 都有一个钥匙列表 rooms[i]，每个钥匙 rooms[i][j] 由 [0,1，...，N-1] 中的一个整数表示，其中 N = rooms[i].length。 钥匙 rooms[i][j] = v 可以打开编号为 v 的房间。

最初，除 0 号房间外的其余所有房间都被锁住。你可以自由地在房间之间来回走动。如果能进入每个房间返回 true，否则返回 false。

示例 1：

```
输入: [[1],[2],[3],[]]
输出: true
解释:
我们从 0 号房间开始，拿到钥匙 1。
之后我们去 1 号房间，拿到钥匙 2。
然后我们去 2 号房间，拿到钥匙 3。
最后我们去了 3 号房间。
由于我们能够进入每个房间，我们返回 true。
```

示例 2：

```
输入: [[1,3],[3,0,1],[2],[0]]
输出: false
解释: 我们不能进入 2 号房间。
```

提示：

- 1 <= rooms.length <= 1000
- 0 <= rooms[i].length <= 1000
- 所有房间中的钥匙数量总计不超过 3000。

2. 简单实现

很基础的BFS

```
class Solution {
public:
    bool canVisitAllRooms(vector<vector<int>>& rooms) {
        int num = rooms.size();
        vector<bool> visited = vector<bool>(num, false);
        queue<int> q;
        q.push(0);
        visited[0] = true;
        int opened = 1;
        while(!q.empty()){
            int cur = q.front();
            q.pop();
```

```
            vector<int> keys = rooms[cur];
            for(int i = 0; i < keys.size(); i++){
                if(!visited[keys[i]]){
                    q.push(keys[i]);
                    visited[keys[i]] = true;
                    opened++;
                }
            }
        }
        if(opened == num)
            return true;
        else
            return false;
    }
};
```