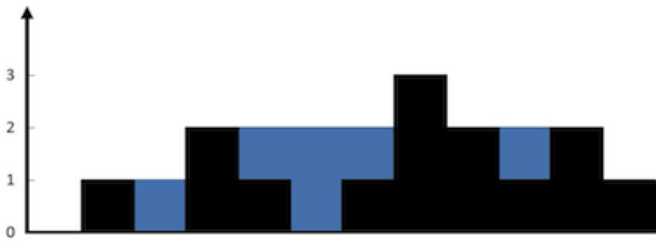
4 5 10 11 22 25 32 33 41 42 44 72 73 81 84 85 87 89 99 123 128 134

1. 栈

1.1. [42] Trapping Rain Water

题目

有一个数组表示方块的高度，计算这些方块最多能存储多少水，要求时间复杂度为`O(n)`，空间复杂度为`O(1)`



思路

- 单调栈，即栈中保存的都是单调的元素，若遇到不单调的元素，则需要进行处理
- 使用递减栈，记下一个入栈元素为`next`
  1. 当`next`小于栈顶时直接入栈
  2. 当`next`大于栈顶时，弹出当前栈顶记为`cur`，新的栈顶记为`top`。由于`top`和`cur`之间的元素都被`cur`入栈之前弹出了，所以它们比`cur`要低；而`cur`和`next`之间的元素没有将`cur`弹出，所以它们也比`cur`要低。因此`cur`位置储水的高度由`top`和`next`的更小值决定，将`cur`位置的储水量汇总到结果中
  3. 不断执行2.直至`next`小于栈顶，然后将`next`入栈

代码

```
class Solution {
public:
    int trap(vector<int>& height) {
        int ans = 0;
        stack<int> s;
        for (int i = 0; i < height.size(); ++i) {
            while (s.size() && height[i] >= height[s.top()]) {
                int cur = s.top();
                s.pop();
                if (s.empty())
                    break;
                ans += (min(height[s.top()], height[i]) - height[cur]) * (i -
s.top() - 1);
            }
            s.push(i);
        }
        return ans;
    }
};
```

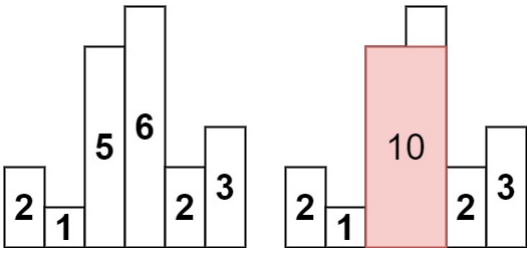
其它

[双指针解法]

1.2. [84] Largest Rectangle in Histogram

题目

有一个数组表示方块的高度，找出其中能围成的面积最大的矩形



思路

使用递增栈，记下一个入栈元素为`next`

1. 当`next`大于栈顶时直接入栈
2. 当`next`小于栈顶时，弹出当前栈顶记为`cur`，新的栈顶记为`top`。由于`top`和`cur`之间的元素都被`cur`入栈之前弹出了，所以它们比`cur`要高；而`cur`和`next`之间的元素没有将`cur`弹出，所以它们也比`cur`要高。因此`top`和`next`之间的元素可以组成高为`cur`的矩形，计算其面积并更新最大面积
3. 不断执行2.直至`next`大于栈顶，然后将`next`入栈

代码

```cpp
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size(), ans = 0;
        heights.push_back(0);
        stack<int> s;
        s.push(-1);
        for (int i = 0; i <= n; ++i) {
            while (s.size() > 1 && heights[i] < heights[s.top()]) {
                int cur = s.top();
                s.pop();
                ans = max(ans, heights[cur] * (i - s.top() - 1));
            }
            s.push(i);
        }
        return ans;
    }
};
```

1.3. [85] Maximal Rectangle

题目

有一个由0和1组成的矩阵，找出其中元素全部为1的最大矩形

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

思路

类似第84题寻找最大的矩形面积，按行遍历矩形，找出遍历到该行时底边在该行上的最大矩形面积并更新结果

代码

```cpp
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty())
            return 0;
        int n = matrix.size(), m = matrix[0].size(), ans = 0;
        vector<int> height(m, 0);
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j)
                height[j] = (matrix[i][j] == '1') ? height[j] + 1 : 0;
            ans = max(ans, largestRectangleArea(height));
        }
        return ans;
    }

    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size(), ans = 0;
        heights.push_back(0);
        stack<int> s;
        s.push(-1);
        for (int i = 0; i <= n; ++i) {
            while (s.size() > 1 && heights[i] < heights[s.top()]) {
                int cur = s.top();
                s.pop();
                ans = max(ans, heights[cur] * (--i - s.top()));
            }
            s.push(i);
        }
        return ans;
    }
};
```

其它

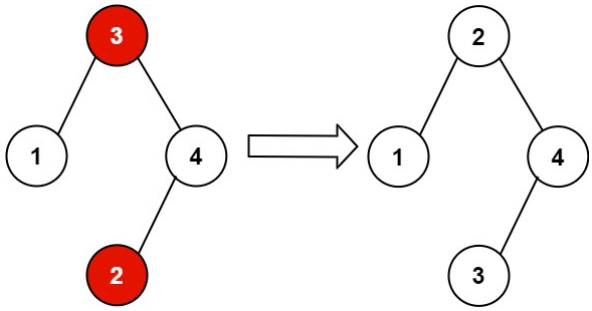
动态规划解法

2. 树

2.1. [99] Recover Binary Search Tree

题目

一棵`BST`中被交换了两个节点，请将该`BST`复原，要求时间复杂度为`O(n)`



思路

- 一开始总是想着根据**局部**父子3个节点的大小关系来找到被交换的节点，但有的时候树的局部总是满足`BST`条件的，只是整体不满足
- 其实只需要进行一次中序遍历，将序列降至一维就能很快地找到失序的两个节点，把它俩记录下来最后交换回去即可

代码

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def recoverTree(self, root: TreeNode) -> None:
        self.first = None
        self.second = None
        self.pre = TreeNode(-1 << 32)
        self.in_order(root)
        self.first.val, self.second.val = self.second.val, self.first.val

    # 寻找中序遍历中顺序异常的节点
    def in_order(self, root):
        if root is None:
            return
        self.in_order(root.left)
        if root.val < self.pre.val:
            if self.first is None:
                self.first = self.pre
            # 不能写else，上一行已经改变first，需要重新判断
            if self.first is not None:
                self.second = root
        self.pre = root
        self.in_order(root.right)

```

3. 字符串

3.1. [5] Longest Palindromic Substring

题目

寻找字符串`s`的最长回文子串

思路

Manacher算法：`O(n)`

- 给`s`的**两端**及每两个字符之间添加一个非法字符`#`得到`ss`，此时`ss`的长度必为奇数。设`ss`以`i`为中心的最长回文子串的最右端为`r`，记录`len[i]=r-i`，表示该最长回文子串在`s`中的长度。设遍历到位置`i`时，`right`为以`[0,i)`为中心的最长回文子串中最右端的位置，`middle`为该最长回文子串的中心：
  1. 当`i`≤`right`时，`i`在`middle`和`right`之间，取`i`关于`middle`对称的点`j`：当`len[j]`≤`right-i`时，以`j`为中心的最长回文子串在以`middle`为中心的最长回文子串的内部，由对称性有`len[i]=len[j]`；当`len[j]>right-i`时，以`i`为中心的最长回文子串会延伸到`right`之外，需要从`right+1`开始逐一匹配，并更新`len[i]`、`middle`和`right`
  2. 当`i>right`时，以i为中心的最长回文子串还没开始匹配，需要从`i+1`开始逐一匹配，并更新`len[i]`、`middle`和`right`
- 由于只需要遍历一遍字符串，因此时间复杂度为`O(n)`

代码

```cpp
class Solution {
public:
    string longestPalindrome(string s) {
        string ss = "#";
        for (int i = 0; i < s.size(); ++i)
            ss = ss + s.at(i) + '#';
        int *len = new int[ss.size()], middle = 0, right = 0;
        len[0] = 0;
        for (int i = 1; i < ss.size(); ++i) {
            len[i] = (i <= right) ? min(len[2 * middle - i], right - i) : 0;
            // 往外延伸
            for (int l = i - len[i] - 1, r = i + len[i] + 1; l >= 0 && r < ss.size() && ss[l] == ss[r]; --l, ++r)
                ++len[i];
            // 更新middle和right
            if (i + len[i] > right) {
                right = i + len[i];
                middle = i;
            }
        }
        int ans = 0;
        for (int i = 1; i < ss.size(); ++i)
            if (len[i] > len[ans])
                ans = i;
        // 注意ss是在每两个字符之间添加了#符号的
        return s.substr((ans - len[ans] + 1) / 2, len[ans]);
    }
};
```

其它

1. 确定回文串中心，再往外不断扩展，时间复杂度为`O(n)`
2. 记`dp[i][j]`表示从`i`到`j`是否构成回文串，`dp[i][j]`可由`dp[i+1][j-1]`得来，时间复杂度为`O(n)`

3.2. [44] Wildcard Matching

题目

给定字符串`s`和正则表达式`p`，判断`s`是否满足`p`。`s`只包含小写字母，`p`中还有特殊字符`?`和`*`

1. `?`表示匹配任意1个字符
2. `*`表示匹配0个或任意个字符

思路

逐一匹配字符，遇到通配符`*`时记录`s`和`p`当前匹配到的位置(只需要记录当前最后一个通配符即可)，当无法匹配时回到记录的通配符位置，同时`s`的匹配位置后移一位表示通配符多匹配一个字符

代码

```cpp
class Solution {
public:
    bool isMatch(string s, string p) {
        int i = 0, j = 0, star_i = -1, star_j = -1;
        while (i < s.size()) {
            if (j < p.size() && (p[j] == '?' || p[j] == s[i]))
                // 精确匹配
                ++i, ++j;
            else if (j < p.size() && p[j] == '*')
                // 保存最近的一个*通配符，先匹配0个字符，后面回退的时候依次增加匹配字符
                star_i = i + 1, star_j = ++j;
            else if (star_j != -1)
                // 未匹配(包括p已经到末尾但s还有未匹配字符)有通配符，回退到最近的*通配符
                i = star_i++, j = star_j;
            else
                // 未匹配(包括p已经到末尾但s还有未匹配字符)也没有通配符，匹配失败
                return false;
        }
        // s已到末尾p可能还有未匹配字符，要匹配剩下的必须是*通配符
        while (j < p.size() && p[j] == '*')
            ++j;
        return j == p.size();
    }
};
```

其它

类似于第10题，同样可以用动态规划求解

4. 查找

4.1. [4] Median of Two Sorted Arrays

题目

找出升序排列的两个数组`nums1`和`nums2`的中位数(总长度为偶数时取中间两个数的平均值)，要求时间复杂度为`O(log(m+n))`，其中`m`和`n`分别为`nums1`和`nums2`的长度

思路

- 看到`log`复杂度联想到二分查找
- 假设中位数将`nums1`和`nums2`分别划分为左右两半部分，`nums1`和`nums2`的左半部分长度分别为`i`和`j`，则`i+j=(m+n)/2`(总长度为奇数时)，`j=(m+n)/2-i`，因此可以二分查找`i`

代码

```cpp
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        // 短的左半部分有i个元素，长的左半部分有j个元素
        if (nums1.size() > nums2.size())
            swap(nums1, nums2);
        // i+j=(m+n+1)/2，故j=(m+n+1)/2-i，二分查找i
        int m = nums1.size(), n = nums2.size();
        int l = 0, r = m, i, j;
        while (l <= r) {
            i = (l + r) / 2, j = (m + n + 1) / 2 - i;
            if (i != 0 && j != n && nums1[i - 1] > nums2[j])
                r = i - 1;
            else if (i != m && j != 0 && nums2[j - 1] > nums1[i])
                l = i + 1;
            else {
                // 找到了切分点，分总元素个数奇偶情况
                int one = (i == 0) ? nums2[j - 1] : ((j == 0) ? nums1[i - 1] :
max(nums1[i - 1], nums2[j - 1]));
                // 这个判断不能放在下一行的后面，因为总共只有1个数时j会越界！！！
                if ((m + n) % 2 == 1)
                    return one;
                int two = (i == m) ? nums2[j] : ((j == n) ? nums1[i] :
min(nums1[i], nums2[j]));
                return (one + two) / 2.0;
            }
        }
        return 0;
    }
};
```

4.2. [33] Search in Rotated Sorted Array

题目

元素各不相同的升序数组被循环右移了`k`个位置，要求在`O(logn)`时间内找出元素在数组中的索引，不存在则返回`-1`

思路

参考二分查找顺序数组的思路，只不过在判断该往左半部分还是右半部分中查找时的逻辑要复杂一些

代码

```python
class Solution:
    def search(self, nums: list[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            if nums[mid] == target:
                return mid
            if target < nums[mid] < nums[right] or nums[mid] < nums[right] < target or nums[right] < target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        return -1
```

4.3. [41] First Missing Positive

题目

有一个未排序的整数数组，找出最小的没有出现的*正*整数，要求时间复杂度为`O(n)`，空间复杂度为`O(1)`

思路

遍历数组，不断将当前元素放回至其正确的位置(如5的正确位置为4)，最后再遍历一次，哪个位置的数不对应，就是没有出现的最小正整数

代码

```cpp
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n; ++i)
            while (nums[i] > 0 && nums[i] <= n && nums[i] != nums[nums[i] - 1])
                swap(nums[i], nums[nums[i] - 1]);
        for (int i = 0; i < n; ++i)
            if (nums[i] != i + 1)
                return i + 1;
        return n + 1;
    }
};
```

4.4. [81] Search in Rotated Sorted Array II

题目

类似第33题，只是被右移的数组中可能含有重复的元素

思路

- 与第33题相比，由于数组首尾可能包含相同的元素，因此无法通过元素大小判断应该往左半部分还是右半部分进行查找
- 先将尾部中与首部相同的元素去除(只有第一次查找的时候才有可能出现首尾元素相同，缩小范围进一步查找的时候不可能出现)，再根据`target`与`arr[mid]`的关系确定往哪半部分继续查找
    1. 右移之后的数组可以分为两组，记第一组为`F`，第二组为`S`(有可能为空)。与首元素比较大小，分别判断`target`和`arr[mid]`位于`F`还是`S`
    2. 若`target`和`arr[mid]`位于不同组，则可以直接判断往哪半部分深入
    3. 若`target`和`arr[mid]`位于相同组，则需要进一步根据`target`和`arr[mid]`的大小判断往哪半部分深入(和有序数组的二分查找一样)
- 第33题的时间复杂度为`O(logn)`，而这里首位有可能存在大量的重复元素，最坏情况下时间复杂度为`O(n)`

代码

```cpp
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1, mid;
        while (right > 0 && nums[right] == nums[0])
            --right;
        while (left <= right) {
            mid = (left + right) / 2;
            if (nums[mid] == target)
                return true;
            bool target_in_left = target >= nums[left];
            bool mid_in_left = nums[mid] >= nums[left];
            // target和mid在不同的部分
            if (target_in_left ^ mid_in_left)
                target_in_left ? right = mid - 1 : left = mid + 1;
            else
                nums[mid] < target ? left = mid + 1 : right = mid - 1;
        }
        return false;
    }
};
```

4.5. [128] Longest Consecutive Sequence

题目

有一个整数数组，求其中能组成最长连续数字序列的长度。如`[4,8,3,1,7,2]`最长连续数字序列为`[1,2,3,4]`，长度为4。要求时间复杂度为`O(n)`

思路

哈希表的查找时间为`O(1)`，先寻找最长连续数字序列的起点，再往后不断延伸序列，记录最大的序列长度

代码

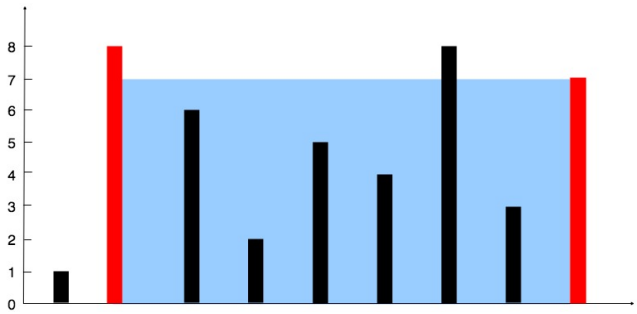
```python
class Solution:
    def longestConsecutive(self, nums: list[int]) -> int:
        nums = set(nums)
        ans = 0
        for n in nums:
            if n - 1 not in nums:
                length = 1
                while n + 1 in nums:
                    length += 1
                    n += 1
                ans = max(ans, length)
        return ans
```

5. 双指针

5.1. [11] Container With Most Water

题目

有一个正整数数组，第`i`个元素代表位置`i`的柱子的长度，求任意两根柱子最多能围成多大的矩形。穷举法当然能够解出来但会超时，能否用`O(n)`的方法？



思路

- 设置两根指针，初始化为最左边和最右边的位置，此时更短的柱子决定了能围成的矩形面积
- 将指针往中间移动，由于底边宽一定会减小，要使得面积增大，高度一定要增加，因此将更短的柱子的指针往中间移动，直至两个指针相遇

代码

```cpp
class Solution {
public:
    int maxArea(vector<int>& height) {
        int l = 0, r = height.size() - 1, res = 0;
        while (l < r) {
            res = max(res, (r - l) * min(height[l], height[r]));
            height[l] < height[r] ? ++l : --r;
        }
        return res;
    }
};
```

5.2. [42] Trapping Rain Water

题目

有一个数组表示方块的高度，计算这些方块最多能存储多少水，要求时间复杂度为`O(n)`，空间复杂度为`O(1)`

思路

- 一个位置的储水量是由其左边和右边最大高度的更小值决定的，最直观的想法是存储每个位置的左右最大高度，空间复杂度为`O(n)`
- 为了减小空间复杂度，可用两个指针分别从两端往中间移动，只要分别记录当前左边和右边的最大高度即可，移动的时候选择高度更低的一端，虽然不知道另一端的最大高度，但最起码比当前端要高，决定储水量的还是当前端的最大高度

代码

```cpp
class Solution {
public:
    int trap(vector<int>& height) {
        int left = 0, right = height.size() - 1, lmax = 0, rmax = 0, ans = 0;
        while (left < right) {
            if (height[left] < height[right]) {
                height[left] > lmax ? lmax = height[left] : ans += lmax - height[left];
                ++left;
            } else {
                height[right] > rmax ? rmax = height[right] : ans += rmax - height[right];
                --right;
            }
        }
        return ans;
    }
};
```

其它

单调栈解法

6. 递归

6.1. [22] Generate Parentheses

题目

产生`n`对括号组成的合法字符串

思路

记第一个左括号及与之对应的右括号组成第1对括号，假设第1对括号之间有`a`对括号，则第1对括号后面有`n-1-a`对括号，遍历`a`将结果组合即可

代码

```cpp
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> ans;
        if (n == 0)
            ans.push_back("");
        else
            for (int i = 0; i < n; ++i)
                for (auto left : generateParenthesis(i))
                    for (auto right : generateParenthesis(n - i - 1))
                        ans.push_back("(" + left + ")" + right);
        return ans;
    }
};
```

6.2. [25] Reverse Nodes in k-Group

题目

将一个链表按每`k`个元素进行逆置，若剩余元素不足`k`个则保留原来的顺序

思路

- 设置两个指针不断地逆置方向，每逆置`k`次就进行一次调整(两组首尾相接处)，这样可行但代码写起来比较复杂
- 用递归的思想，将从第`k+1`个位置开始的链表递归地完成逆置，然后只要调整前`k`个元素即可

代码

```cpp
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* cur = head;
        int cnt = 0;
        while (cur != NULL && cnt != k) {
            cur = cur->next;
            ++cnt;
        }
        if (cnt == k) {
            cur = reverseKGroup(cur, k);
            while (--cnt >= 0) {
                ListNode* tmp = head->next;
                head->next = cur;
                cur = head;
```

```
                head = tmp;
            }
            head = cur;
        }
        return head;
    }
};
```

7. DP

7.1. [10] Regular Expression Matching

题目

给定字符串`s`和正则表达式`p`，判断`s`是否满足`p`。`s`只包含小写字母，`p`中还有特殊字符`.`和`*`，但每个`*`前面至少会有1个字母

1. `.`表示匹配任意1个字符
2. `*`表示匹配0个或任意个*前导字符*

思路

- 很明显可以一个个字符进行匹配，不好搞的是`*`，需要用`DFS`依次试探任意个字符，但直接`DFS`肯定超时。`DFS`超时一般都是因为存在大量的重复计算，因此可以转化为记忆`DFS`或`DP`
- 设`dp[i][j]`表示`s`的前`i`个字符是否满足`p`的前`j`个字符
    1. 当`p[j-1]!=*`时，`s[i-1]`和`p[j-1]`必须匹配
    2. 当`p[j-1]==*`时，这个`*`可以匹配0次即`dp[i][j-2]`，也可以匹配多次即`s[i-1]`和`p[j-2]`必须匹配

代码

```
class Solution {
public:
    bool isMatch(string s, string p) {
        bool dp[25][35] = { false };
        // i==0时只有全是通配符的才匹配
        dp[0][0] = true;
        for (int j = 2; j <= p.size(); j += 2)
            dp[0][j] = dp[0][j - 2] && p[j - 1] == '*';
        // j==0时只有i==0才匹配(上述已处理)
        // j==1时只有i==1且两个字符匹配时才匹配
        dp[1][1] = p[0] == '.' || p[0] == s[0];
        for (int i = 1; i <= s.size(); ++i)
            for (int j = 2; j <= p.size(); ++j) {
                if (p[j - 1] != '*')
                    // 只匹配一个字符
                    dp[i][j] = dp[i - 1][j - 1] && (p[j - 1] == '.' || p[j - 1] == s[i - 1]);
                else
                    // 通配符
                    dp[i][j] = dp[i][j - 2] || (dp[i - 1][j] && (p[j - 2] == '.' || p[j - 2] == s[i - 1]));
```

```
            }
        return dp[s.size()][p.size()];
    }
};
```

7.2. [32] Longest Valid Parentheses

题目

有一个由`(`和`)`组成的字符串`s`，找出其中合法括号子串的最大长度

思路

设`dp[i]`表示以`s[i]`为结尾的合法括号子串的最大长度

1. 当`s[i]=='('`时，`dp[i]=0`
2. 当`s[i]==')'`且`s[i-1]=='('`时，`dp[i]=dp[i-2]+2`
3. 当`s[i]==')'`且`s[i-1]!='('`时，找到`s[i-dp[i-1]-1]`即与之匹配的位置，如果其为`(`则可匹配，记得还要加上再往前一个位置的`dp[i-dp[i-1]-2]`

代码

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int *dp = new int[s.size()](), ans = 0;
        for (int i = 1; i < s.size(); ++i)
            if (s[i] == ')') {
                if (s[i - 1] == '(')
                    dp[i] = (i > 1 ? dp[i - 2] : 0) + 2;
                else if (i - dp[i - 1] > 0 && s[i - dp[i - 1] - 1] == '(')
                    dp[i] = dp[i - 1] + 2 + (i - dp[i - 1] > 1 ? dp[i - dp[i - 1] - 2] : 0);
                ans = max(ans, dp[i]);
            }
        return ans;
    }
};
```

其它

也可用计数的方法，记录`(`和`)`的个数，若相等则找到一个合法子串，若`(`的个数小于`)`则必定非法，使用两个指针指定当前子串，从左往右和**从右往左**依次扫描一遍并记录最大长度

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int start = 0, end = 0, n = s.size(), left = 0, right = 0, ans = 0;
        while (end < n) {
            s[end] == '(' ? ++left : ++right;
            ++end;
            if (left == right)
                ans = max(ans, end - start);
            else if (left < right)
                left = right = 0, start = end;
```

```
        }
        start = end = n - 1, left = right = 0;
        while (start >= 0) {
            s[start] == '(' ? ++left : ++right;
            --start;
            if (left == right)
                ans = max(ans, end - start);
            else if (left > right)
                left = right = 0, end = start;
        }
        return ans;
    }
};
```

7.3. [72] Edit Distance

题目

给定两个字符串，求将一个字符串`s`转化为另一个字符串`t`需要操作的最少次数，操作有以下3种：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

思路

不要局限在考虑在哪个位置插入、删除或替换，设`dp[i][j]`表示`s`的前`i`个字符转化为`t`的前`j`个字符需要操作的最少次数

1. 当`s[i-1]==t[j-1]`时，`dp[i][j]=dp[i-1][j-1]`
2. 当`s[i-1]!=t[j-1]`时，`dp[i][j]`可由`dp[i-1][j]`插入一个字符、可由`dp[i][j-1]`删除一个字符、或由`dp[i-1][j-1]`替换一个字符得来，取其中的最小值

代码

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int n = word1.size(), m = word2.size();
        vector<int> dp(m + 1, 0);
        for (int i = 1; i <= m; ++i)
            // 初始值dp[0][i]=i
            dp[i] = i;
        for (int i = 1; i <= n; ++i) {
            int pre = dp[0];
            // 初始值dp[i][0]=i
            dp[0] = i;
            for (int j = 1; j <= m; ++j) {
                int cur = dp[j];
                if (word1[i - 1] == word2[j - 1])
                    dp[j] = pre;
                else
                    dp[j] = min(min(pre, cur), dp[j - 1]) + 1;
                pre = cur;
            }
        }
```

```
        return dp[m];
    }
};
```

7.4. [85] Maximal Rectangle

题目

有一个由0和1组成的矩阵，找出其中元素全部为1的最大矩形

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

思路

遍历每一元素，遍历到第`i`行第`j`列时，计算以元素`(i,j)`为右下角的最大矩形面积并更新结果。期间记录3个数组：`height`表示该列向上移动包含1的最大高度即矩形的高，`left`表示矩形的左边界，`right`表示矩形的右边界，`left`和`right`共同决定了矩形的宽

1. 如果`arr[i][j]==1`，`height[i][j]=height[i-1][j]+1`，否则`height[i][j]=0`
2. 记录该行向左移动包含1的最左边位置`cur_left`，`left[i][j]`由`cur_left`和`left[i-1][j]`共同决定
3. `right[i][j]`的更新规则同理

代码

```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty())
            return 0;
        int n = matrix.size(), m = matrix[0].size(), ans = 0;
        vector<int> height(m, 0), left(m, 0), right(m, m - 1);
        for (int i = 0; i < n; ++i) {
            int cur_left = 0, cur_right = m - 1;
            for (int j = 0; j < m; ++j) {
                if (matrix[i][j] == '1') {
                    ++height[j];
                    left[j] = max(left[j], cur_left);
                } else {
                    height[j] = 0;
                    cur_left = j + 1;
                    // 复原left，以便下一行left的计算，同时因为height为0了所以不影响面
积的计算
                    left[j] = 0;
                }
            }
            for (int j = m - 1; j >= 0; --j) {
                if (matrix[i][j] == '1')
                    right[j] = min(right[j], cur_right);
                else {
```

```
                    cur_right = j - 1;
                    // 同样，复原right
                    right[j] = m - 1;
                }
                ans = max(ans, (right[j] - left[j] + 1) * height[j]);
            }
        }
        return ans;
    }
};
```

其它

[单调栈解法]

7.5. [87] Scramble String

题目

有一种针对字符串 `s` 的操作 `A` 过程如下：

1. 如果 `s` 的长度大于1，则将 `s` 切分为两个非空子串 `x` 和 `y`，随机选择是否交换 `x` 和 `y` 的顺序，并且对 `x` 和 `y` 也递归地做 `A` 操作
2. 如果 `s` 的长度等于1，则停止操作

给定两个字符串 `s` 和 `t`，判断 `s` 和 `t` 是否可由 `A` 操作互相得到

思路

- 按照 `A` 操作的思路，对每一个切分点都做一次尝试，将 `s` 切分为 `s1` 和 `s2`、`t` 切分为 `t1` 和 `t2`，递归地判断两对子串是否可由 `A` 操作互相得到
- 如果直接递归计算的话会存在大量重复的计算，因此可以使用记忆数组保存已经做过的计算

  设 `mem[i][j][k]` 表示 `s` 起点为 `i` 长度为 `k` 的子串和 `t` 起点为 `j` 长度为 `k` 的子串是否可由 `A` 操作互相得到，`1` 表示可以、`-1` 表示不可以、`0` 表示还不确定
- 自上而下的记忆可以换成自下而上的DP

代码

```
class Solution {
public:
    int mem[31][31][31] = { 0 };
    bool isScramble(string s1, string s2) { return dfs(s1, s2, 0, 0, s1.size()); }

    bool dfs(string s1, string s2, int l1, int l2, int len) {
        if (mem[l1][l2][len])
            return mem[l1][l2][len] == 1;
        if (len == 1)
            return s1[l1] == s2[l2];
        for (int i = 1; i < len; ++i)
            if ((dfs(s1, s2, l1, l2, i) && dfs(s1, s2, l1 + i, l2 + i, len - i))
                || (dfs(s1, s2, l1, l2 + len - i, i) && dfs(s1, s2, l1 + i, l2,
len - i))) {
                mem[l1][l2][len] = 1;
                return true;
            }
```

```
            mem[l1][l2][len] = -1;
            return false;
        }
};
```

7.6. [123] Best Time to Buy and Sell Stock III

题目

有一个数组表示每天股票的售价，你可以在某天买入股票，然后在另一天卖出。有几个要求：

1. 不能在同一天买入和卖出
2. 最多交易2笔
3. 买入之前必须将之前的卖出

计算能获得的最大利润

思路

- 记`buy1[i]`、`sell1[i]`、`buy2[i]`、`sell2[i]`分别表示第`i`天买入第1笔时、卖出第1笔时、买入第2笔时、卖出第2笔时的最大利润。状态转移方程如下

$$\begin{cases} buy1[i] = \max\{buy1[i-1], -prices[i]\} \\ sell1[i] = \max\{sell1[i-1], buy1[i-1] + prices[i]\} \\ buy2[i] = \max\{buy2[i-1], sell1[i-1] - prices[i]\} \\ sell1[i] = \max\{sell1[i-1], buy2[i-1] + prices[i]\} \end{cases}$$

- 由于是最多交易2笔而不是必须交易2笔，因此可以允许在同一天买入和卖出，因为这样的交易不会影响最终的利润。状态可以简化为

$$\begin{cases} buy1 = \max\{buy1, -prices[i]\} \\ sell1 = \max\{sell1, buy1 + prices[i]\} \\ buy2 = \max\{buy2, sell1 - prices[i]\} \\ sell1 = \max\{sell1, buy2 + prices[i]\} \end{cases}$$

- 初始值，考虑第0天的情况，买入第1笔`buy1=-prices[0]`，再卖出第1笔`sell1=0`，由于允许交易多笔而不影响结果，同理可以计算`buy2=-prices[0]`和`sell2=0`。最终答案为`sell2`，卖出多笔的利润不会比卖出更少笔的利润低

代码

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        buy1, sell1, buy2, sell2 = -prices[0], 0, -prices[0], 0
        for p in prices[1:]:
            buy1 = max(buy1, -p)
            sell1 = max(sell1, buy1 + p)
            buy2 = max(buy2, sell1 - p)
            sell2 = max(sell2, buy2 + p)
        return sell2
```

其它

这题的状态设计不好想，若扩展至一般情况：最多只能交易`k`笔，可以设`dp[i][j]`表示第`i`天交易进行至`j`状态时的最大利润(`j`取0到`2k-1`，表示第`j/2`笔交易，`j%2==0`表示买入，`j%2==1`表示卖出)，状态转移方程类似，也可简化只保留`j`维度

8. 未分类

8.1. [73] Set Matrix Zeroes

题目

有一个矩阵，如果某个元素为0，则将该行和该列的所有元素都置为0，要求空间复杂度为`O(1)`

思路

- 直观的想法是另外创建一个大小一样的矩阵，将元素为0的行和列置为0，再填充剩余元素。要就地设置，却不能直接填0，因为同一行(列)有可能有多个0，这样处理第一个0之后就无法确定后面的0是填充的还是原来就有的
- 只需要标记每行和每列是否需要填充0即可，这样的话标记一共有`m+n`个，空间复杂度依然不是`O(1)`。可以单独拿出一行和一列来作为标记(如第0行和第0列)，元素`(0,0)`记录第0行的标记，第0列的标记用另一个变量记录，这也是唯一的空间消耗

代码

```python
class Solution:
    def setZeroes(self, matrix: list[list[int]]) -> None:
        m, n = len(matrix), len(matrix[0])
        first_col_zero = 1
        for i in range(m):
            if matrix[i][0] == 0:
                first_col_zero = 0
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0
                    matrix[0][j] = 0
        # 填充除标记以外的元素
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0
        # 填充第0行
        if matrix[0][0] == 0:
            for j in range(n):
                matrix[0][j] = 0
        # 填充第0列
        if first_col_zero == 0:
            for i in range(m):
                matrix[i][0] = 0
```

8.2. [89] Gray Code

题目

`n`位二进制数共2^n个，将其排列成一个数组，使得相邻两个数的码距为1，第一个数和最后一个数的码距也为1，返回任意符合要求的数组

思路

可以直接由`n-1`位二进制的答案得到`n`位二进制的答案，如2位二进制的一个结果为`(00,01,11,10)`，在其开头分别加上0和1可以得到3位二进制的两组数`(000,001,011,010)`和`(100,101,111,110)`，这样可以保证两组数组内相邻数的码距都为1，将第二组数逆置得到`(110,111,101,100)`接在第一组数后面即可得到答案

代码

```cpp
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> ans;
        ans.push_back(0);
        for (int i = 0; i < n; ++i)
            for (int j = ans.size() - 1; j >= 0; --j)
                ans.push_back(ans[j] ^ (1 << i));
        return ans;
    }
};
```

其它

当然可以直接进行搜索，每次将变换一个二进制位且还未出现过的数加入数组中

8.3. [134] Gas Station

题目

一段环形的公路有`n`个加油站，`gas`数组存放每个加油站的油量，`cost`数组存放每个加油站到达下一个加油站所需要的油量。找出起始加油站，使得可以成功绕公路一圈，如果不存在则返回-1。要求时间复杂度为`O(n)`

思路

假设从`A`出发无法到达`B`，那么从`A`和`B`之间的任意站点出发也无法到达`B`。这样最多遍历两次就可以找出答案

代码

```cpp
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size(), start, total_gas = -1, total_cost = 0;
        for (int i = 0; i < 2 * n; ++i) {
            if (total_gas < total_cost)
                total_gas = total_cost = 0, start = i % n;
            else if (i % n == start)
                return start;
            total_gas += gas[i % n], total_cost += cost[i % n];
        }
        return -1;
    }
};
```