数组

二分法

704. 二分查找：外面是一层while循环，左闭右开。注意`middle = left + (right - left) / 2`这种写法（如果直接相加小心超过int上限

双指针法

27. 移除元素：一开始是重合的，如果fast指针遇到了要移除的元素就先走
28. 有序数组的平方：从两头往里比较，`A[i] * A[i] < A[j] * A[j]`那么`result[k--] = A[j] * A[j];`
29. 三数之和：先固定第一个数，第二个数left为`i+1`，第三个数right从末尾往左
30. 四数之和：先把前两个数加上，然后继续left & right往中间移动，和三数之和一样
31. 反转字符串
32. 反转字符串II
33. 回文子串：从中间往两边开始检查

```
        for i in range(len(s)):
            result += self.extend(s, i, i, len(s)) #以i为中心
            result += self.extend(s, i, i+1, len(s)) #以i和i+1为中心
```

滑动窗口

209. 长度最小的子数组

前缀和

区间和：可以用一个额外数组`p[i]`表示`vec[i]`从下标为0一直加到i的和。这样求区间和可以直接做减法。

其他

59. 螺旋矩阵II：没技巧，纯靠算。`loop, mid = n // 2, n // 2`，`loop`在最外层控制循环次数。
60. 开发商购买土地：行末尾统计的是列划分，列末尾统计的是行划分

链表

```
// 单链表
struct ListNode {
    int val;  // 节点上存储的元素
    ListNode *next;  // 指向下一个节点的指针
    ListNode(int x) : val(x), next(NULL) {}  // 节点的构造函数
};
// 通过自己定义的构造函数初始化
ListNode* head = new ListNode(5);
// 使用默认构造函数初始化
ListNode* head = new ListNode();
head->val = 5;
```

虚拟头节点

203. 移除链表元素：不是很需要双指针，判断`while(curr.next)`就行

204. 设计链表

205. 反转链表

206. 两两交换链表中的节点

207. 删除链表的倒数第N个节点：fast先走n+1，然后判断是否为null更方便

02.07. 链表相交

其他

142. 环形链表II：`x = (n - 1) (y + z) + z`。当 n为1的时候，`x = z`。即，快慢指针相遇后，让一个指针从头节点出发，让它和快慢指针同时走，再相遇就是环的入口结点。

哈希表

242. 有效的字母异位词

243. 两个数组的交集

244. 快乐数：会重复

245. 两数之和：遍历过的就加入集合，然后在里面找`complement = target - num`即可。

246. 四数相加II

247. 赎金信

字符串

> 注意三个函数:
>
> 而且注意，python里的string是不可变的。如果要变，就要创建一个新的对象。
>
> `join()`：用指定字符链接
>
> `split()`：用指定字符分割（默认为所有）
>
> `strip()`：只去除前后
>
> `reversed()`用于字符串，但是会返回迭代器，需要用`join`连接

> `reverse()`用于list

344. 反转字符串

345. 反转字符串II

346. 翻转字符串里的单词

KMP

```
    def getNext(self, next: List[int], s: str) -> None:
        j = 0
        next[0] = 0
        for i in range(1, len(s)):
            while j > 0 and s[i] != s[j]:
                j = next[j - 1]
            if s[i] == s[j]:
                j += 1
            next[i] = j
```

28. 实现 strStr()：我觉得难。实在不行就用find吧。

29. 重复的子字符串：判断条件是`if nxt[-1] != 0 and len(s) % (len(s) - nxt[-1]) == 0:`

栈与队列

232. 用栈实现队列

233. 用队列实现栈：可以优化只使用一个队列

234. 有效的括号

235. 删除字符串中的所有相邻重复项

236. 逆波兰表达式求值

237. 滑动窗口最大值：单调队列

238. 前K个高频元素：优先级队列，`heapq.heappush(pri_que, (freq, key))`。第一个元素freq为排序的索引

二叉树

> 完全二叉树：如果有值，必然在左侧
>
> 二叉搜索树：左<中<右
>
> 平衡二叉搜索树AVL

递归遍历/迭代遍历(栈，中结点后push(NULL))：

144. 二叉树的前序遍历

145. 二叉树的中序遍历

146. 二叉树的后序遍历

层序遍历（队列）：

102. 二叉树的层序遍历

103. 二叉树的层次遍历 II

104. 二叉树的右视图

105. 二叉树的层平均值

106. N叉树的层序遍历

107. 在每个树行中找最大值

108. 填充每个节点的下一个右侧节点指针

109. 填充每个节点的下一个右侧节点指针II

110. 二叉树的最大深度

111. 二叉树的最小深度

101. 对称二叉树：使用递归

102. 完全二叉树的节点个数

103. 平衡二叉树

104. 二叉树的所有路径：需要回溯

105. 左叶子之和

106. 找树左下角的值：层序

107. 路径总和

108. 路径总和 II

109. **从前序与中序遍历序列构造二叉树**

110. **从中序与后序遍历序列构造二叉树**

111. 最大二叉树

112. 合并二叉树

113. 二叉搜索树中的搜索

114. 验证二叉搜索树：中序遍历后加入数组，然后判定是否是升序数组

115. 二叉搜索树的最小绝对差：中序

116. 二叉搜索树中的众数

117. 二叉树的最近公共祖先

118. 二叉搜索树的最近公共祖先：利用其性质，如果`p < root.val < q`，可以直接返回`root`了

119. 二叉搜索树中的插入操作

120. 删除二叉搜索树中的节点：最关键的操作，把值为key的结点和右孩子的最左侧孩子互换

121. 修剪二叉搜索树：有点tricky

122. 将有序数组转换为二叉搜索树：找到中点，再分别递归左右

123. 把二叉搜索树转换为累加树：中序遍历，但从右边开始版

回溯

```
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
        处理节点;
        backtracking(路径，选择列表); // 递归
        回溯，撤销处理结果 // 可能不止一个pop，还有其他需要撤销的
    }
}
```

剪枝优化：**如果for循环选择的起始位置之后的元素个数 已经不足 我们需要的元素个数了，那么就没有必要搜索了。**

无放回组合：**一个**集合，需要`startIndex`

77. 无列：问题
78. 组合总和III
79. 分割回文串
80. 复原IP地址

无放回有重复：

78. 组合总和II：有重复元素，sort后去重（和前一个一样直接跳过）
79. 子集II：如前，直接跳过。也可以用`used`去重。
80. 递增子序列：不可排序，需要使用`set()`对本层使用过的节点去重（横向去重）

多个集合互不影响，不需要`startIndex`

77. 电话号码的字母组合

有放回组合：

39. 组合总和：`self.backtracking(candidates, target, total, i, path, result)  # 不用i+1了，表示可以重复读取当前的数`

排列：不用`startIndex`，但要用`used`数组去重

46. 全排列：不重复
47. 全排列 II：重复。`if (i > 0 and nums[i] == nums[i - 1] and not used[i - 1]) or used[i]: continue`

推荐用used，复杂度更低。

332. 重新安排行程
333. N皇后：`chessboard = ['.' * n for _ in range(n)]`这样用棋盘。递归row，while col，在while里检查is_valid，确认后再递归。
334. 解数独：行列和数全用for来循环了，然后就一个`self.backtracking(board)`。就是return 为bool。

贪心

455. 分发饼干：小饼干优先
456. 摆动序列：**删除单调坡度上的节点，那么这个坡度就可以有两个局部峰值。**考虑平坡情况：上下坡，首位两端，单调坡。
`if (preDiff <= 0 and curDiff > 0) or (preDiff >= 0 and curDiff < 0):`
也可以用dp做，但是复杂度更高
457. 最大子序和
458. 买卖股票的最佳时机 II：多次买卖
459. 跳跃游戏
460. K次取反后最大化的数组和：按abs降序
461. 加油站：**前累加rest[i]的和curSum一旦小于0，起始位置至少要是i+1**
462. 分发糖果：从前向后和从后向前
463. 柠檬水找零：优先给大钱
464. 根据身高重建队列：类似分发糖果，两个维度一定要先按一个维度来排列。**优先按身高高的people的k来插入。插入操作过后的people满足队列属性**
`people.sort(key=lambda x: (-x[0], x[1]))`，然后从前往后遍历`insert`
465. 用最少数量的箭引爆气球：`points[i][1] = min(points[i - 1][1], points[i][1]) # 更新重叠气球最小右边界`
466. 无重叠区间：也可以算无重叠区间的数量，然后减完就是要消除的数量
467. 划分字母区间：先遍历一遍，找到`last_occurance`下标；再遍历一遍，如果与index相同则说明到了最远边界了，可以划分
468. 合并区间
469. 单调递增的数字：从后往前遍历
470. 监控二叉树：三种情况，列清楚就好了。从下往上遍历（后序遍历）

动态规划

总结：子序列/背包一般都是用dp

一般来说`dp = [0] * (n + 1)`

方法数：`dp[i] = dp[i - 1] + dp[i - 2]`

最大/最小花费：`min() or max()`

递归的拆分：`i`里面再遍历`j`，比如514整数拆分和96不同二叉搜索树

背包：`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i])`

背包问题：使用二维数组，`i`为物品，`j`为背包重量，`dp`为价值。

```
dp = [[0] * (bagweight + 1) for _ in range(n)]
# 初始化第一行
for j in range(weight[0], bagweight + 1):
    dp[0][j] = value[0]

for i in range(1, n):
    for j in range(bagweight + 1):
        if j < weight[i]:
            dp[i][j] = dp[i - 1][j]
        else:
            dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i])

print(dp[n - 1][bagweight])
```

```
# 状态压缩
for i in range(n):  # 应该先遍历物品，如果遍历背包容量放在上一层，那么每个dp[j]就只会放入一个物品
    for j in range(bagweight, weight[i]-1, -1):  # 倒序遍历背包容量是为了保证物品i只被放入一次
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i])
```

- 01背包：只能选一次，`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);`
  - 状态压缩：第二层循环倒序遍历
- 完全背包：可选好几次
  - 状态压缩：第二层可以正序，初始化时`dp[0] = 1`
    - 组合：先物品后背包
    - 排列：先背包后物品
- 多重背包：不同物品数量不同。但把Mi件摊开，其实就是一个01背包问题。或者把个数放在最内层：

```
            for (int k = 1; k <= nums[i] && (j - k * weight[i]) >= 0; k++) {  // 遍历个数
                dp[j] = max(dp[j], dp[j - k * weight[i]] + k * value[i]);
            }
```

⚠️其他的dp都是`dp[i]=xxx`，而背包问题是`dp[j]`

编辑距离：

- 删除：`dp[i][j] = min({dp[i - 1][j - 1] + 2, dp[i - 1][j] + 1, dp[i][j - 1] + 1})`
- 可替换：`dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]}) + 1;`

509. 斐波那契数：方法数

510. 爬楼梯：方法数

511. 使用最小花费爬楼梯

512. 不同路径：方法数

513. 不同路径 II：障碍物直接跳过处理就好了，不需要特殊判断。且初始化遇到障碍物后，后面的全是0

514. 整数拆分：`dp[i] = max(dp[i], dp[i - j] * dp[j])`

515. 不同的二叉搜索树：`dp[i] += dp[j - 1] * dp[i - j]`

516. 单词拆分：`dp[i]` 表示字符串的前 i 个字符是否可以被拆分成单词；`if dp[j] and s[j:i] in wordSet: dp[i] = True`

01背包

416. 分割等和子集

417. 最后一块石头的重量II：其实就是分割等和子集的变体

418. 目标和：用`target`和`sum`转换一下还是分割等和子集，但是这是求方法数，所以`dp[j] += dp[j - num]`。也可以用回溯法做，但核心都是转换为等和子集求方法数。

419. 一和零：背包容量有两个维度

```
        dp = [[0] * (n + 1) for _ in range(m + 1)]  # 创建二维动态规划数组，初始化为0
        # 遍历物品
        for s in strs:
            ones = s.count('1')  # 统计字符串中1的个数
            zeros = s.count('0')  # 统计字符串中0的个数
            # 遍历背包容量且从后向前遍历
            for i in range(m, zeros - 1, -1):
                for j in range(n, ones - 1, -1):
                    dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)  # 状态转移方程
        return dp[m][n]
```

完全背包

518. 零钱兑换II：组合

519. 组合总和 IV：实际上在求排列。

520. 爬楼梯（进阶版）

521. 零钱兑换：初始化时要初始化为最大值。我认为是求组合数。

522. 完全平方数

双（多）状态DP：几个状态第二维数组就多大

198. 打家劫舍：一维DP，相当于爬楼梯

199. 打家劫舍II：连成一个环就是算两次，`nums[:-1]`和`nums[1:]`，其余逻辑和打家劫舍一样。

200. 打家劫舍 III：树形DP，二位数组（**下标0：不偷，下标1：偷**）

```
        left = self.traversal(node.left)
        right = self.traversal(node.right)
        # 不偷当前节点，偷子节点
        val_0 = max(left[0], left[1]) + max(right[0], right[1])
        # 偷当前节点，不偷子节点
        val_1 = node.val + left[0] + right[0]
        return (val_0, val_1)
```

201. 买卖股票的最佳时机：只买卖一次，直接贪心；如果要用DP，那么**下标0持有，下标1不持有**

```
        for i in range(1, length):
            dp[i][0] = max(dp[i-1][0], -prices[i])
            dp[i][1] = max(dp[i-1][1], prices[i] + dp[i-1][0])
```

202. 买卖股票的最佳时机II：多次买卖。121题变成`dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i])`即可。

203. 买卖股票的最佳时机III：两笔交易，4+1个状态。

```
        for i in range(1, len(prices)):
            dp[i][0] = dp[i-1][0] # 可以不要
            dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i]) #如果不要状态0，
这里是max(dp[i-1][1], -prices[i])
            dp[i][2] = max(dp[i-1][2], dp[i-1][1] + prices[i])
            dp[i][3] = max(dp[i-1][3], dp[i-1][2] - prices[i])
            dp[i][4] = max(dp[i-1][4], dp[i-1][3] + prices[i])
```

204. 买卖股票的最佳时机IV：k笔交易，`2*k+1`个状态

```
        for i in range(1, len(prices)):
            dp[i][0] = dp[i-1][0]
            for j in range(1, 2*k+1, 2):
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-1] - prices[i])
                dp[i][j+1] = max(dp[i-1][j+1], dp[i-1][j] + prices[i])
```

205. 最佳买卖股票时机含冷冻期：4个状态，持有/不持有/卖出/冷冻，要画状态机

```
        for i in range(1, n):
            dp[i][0] = max(dp[i-1][0], max(dp[i-1][3], dp[i-1][1]) -
prices[i])  # 当前持有股票的最大利润等于前一天持有股票的最大利润或者前一天不持有股票且不处
于冷冻期的最大利润减去当前股票的价格
            dp[i][1] = max(dp[i-1][1], dp[i-1][3])  # 当前不持有股票且处于冷冻期
的最大利润等于前一天持有股票的最大利润加上当前股票的价格
            dp[i][2] = dp[i-1][0] + prices[i]  # 当前不持有股票且不处于冷冻期的最
大利润等于前一天不持有股票的最大利润或者前一天处于冷冻期的最大利润
            dp[i][3] = dp[i-1][2]  # 当前不持有股票且当天卖出后处于冷冻期的最大利润等
于前一天不持有股票且不处于冷冻期的最大利润
```

206. 买卖股票的最佳时机含手续费：122的状态1减去手续费即可。

以下题目都要写判断条件：

总结：**如果是不连续的，需要用`max()`；如果是连续子数组，那么直接用`=`**。对于字符串，初始化的时候size+1，遍历时都从1开始；非字符串直接用size即可。

300. 最长递增子序列：`p[i] = max(dp[i], dp[j] + 1)`，和单词拆分差不多。注意里面是`dp[i]`而非`dp[i-1]`，因为是通过遍历`j`将`dp[i]`慢慢变大。
301. 最长连续递增序列：直接`dp[i] = dp[i - 1] + 1`即可
302. 最长重复子数组：不需要用max，直接`dp[i][j] = dp[i - 1][j - 1] + 1;`
303. 最长公共子序列：字符串版，初始化的时候size+1

```
if (text1[i - 1] == text2[j - 1]) {
    dp[i][j] = dp[i - 1][j - 1] + 1;
} else {
    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
}
```

304. 不相交的线：和前一题一模一样
305. 最大子序和：用贪心的思路
306. 判断子序列：和最长公共子序列一模一样，但else里其实不用max，直接写成`dp[i][j - 1]`也可。因为`t`必定比`s`长
307. 不同的子序列：注意初始化。

```
        for i in range(1, len(s)+1):
            for j in range(1, len(t)+1):
                if s[i-1] == t[j-1]:
                    dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
                else:
                    dp[i][j] = dp[i-1][j]
```

308. 两个字符串的删除操作

```
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(dp[i-1][j-1] + 2, dp[i-1][j] + 1, dp[i][j-
1] + 1)
```

309. 编辑距离

```
        for i in range(1, len(word1)+1):
            for j in range(1, len(word2)+1):
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1
```

310. 回文子串：找有多少个回文串。这题的dp得用true/false，而且画一下转移矩阵会发现，得从下到上/左到右开始遍历。从两边往中心检查。或者用双指针做，从中心往两边检查。

```
        for i in range(len(s)-1, -1, -1): #注意遍历顺序
            for j in range(i, len(s)):
                if s[i] == s[j] and (j - i <= 1 or dp[i+1][j-1]):
                    result += 1
                    dp[i][j] = True
```

311. 最长回文子序列：找回文串最长多少。

```
        for i in range(len(s)-1, -1, -1):
            for j in range(i+1, len(s)):
                if s[i] == s[j]:
                    dp[i][j] = dp[i+1][j-1] + 2
                else:
                    dp[i][j] = max(dp[i+1][j], dp[i][j-1])
```