贪心算法

真实场景：每一步最好，整体不一定最好

因此需要证明：其成立

可以做为辅助算法：最小生成树、

贪心算法：不回退，每一步（子问题）都找到当前状态下，最好或最优的选择，从而得出全局最好或最优的算法

动态规划：回退，保存之前的运算结果，和当前最好的选择，二者比较，择优录取。

贪心：【每一步做出最优判断】

回溯：【回退：清除数据】

动态规划：【每一步做出最优判断 + 回退 = 二者比较，择优录取】

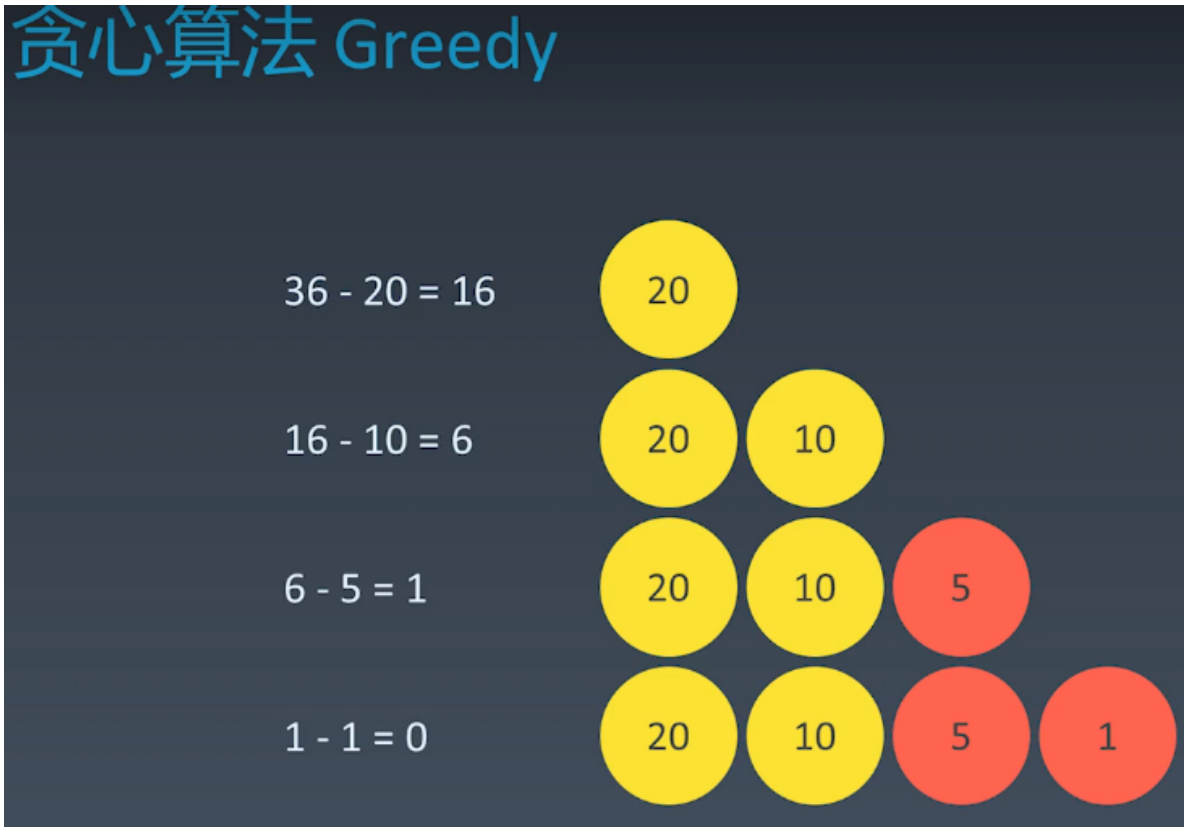
使用场景:

从前向后、从后向前

证明可以用贪心

需要论证成立，需要特殊条件

举例：20、10、5、1 倍数关系，才可以用；否则不行



题目

860. 柠檬水找零

难度简单119

在柠檬水摊上，每一杯柠檬水的售价为 `5` 美元。

顾客排队购买你的产品，（按账单 `bills` 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 `5` 美元、`10` 美元或 `20` 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 `5` 美元。

注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 `true` ，否则返回 `false` 。

示例 1：

```
输入：[5,5,5,10,20]
输出：true
解释：
前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
由于所有客户都得到了正确的找零，所以我们输出 true。
```

示例 2：

```
输入：[5,5,10]
输出：true
```

示例 3：

```
输入：[10,10]
输出：false
```

示例 4：

```
输入：[5,5,10,10,20]
输出：false
解释：
前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
由于不是每位顾客都得到了正确的找零，所以答案是 false。
```

提示：

- `0 <= bills.length <= 10000`
- `bills[i]` 不是 `5` 就是 `10` 或是 `20`

贪心分析

Intuition:

When the customer gives us `$20`, we have two options:

1. To give three `$5` in return
2. To give one `$5` and one `$10`.

On insight is that the second option (if possible) is always better than the first one.
Because two `$5` in hand is always better than one `$10`

Explanation:

Count the number of `$5` and `$10` in hand.

if (customer pays with `$5`) `five++`;
if (customer pays with `$10`) `ten++, five--`;
if (customer pays with `$20`) `ten--, five--` or `five -= 3`;

Check if five is positive, otherwise return false.

Time Complexity

Time `O(N)` for one iteration
Space `O(1)`

代码

Python ***

```python
class Solution:
    def lemonadeChange(self, bills: List[int]) -> bool:
        # time: 160ms, beats 95%
        if not bills: return False

        five = ten = 0

        for bill in bills:
            # 遍历每个账单
            if bill == 5:
                # 收一个5美元，不用找零
                five += 1
            elif bill == 10:
                # 收一个10美元，找零5美元
                five -= 1
                ten += 1
            elif ten > 0:          # 直接写即可
                # 收一个20美元，两种找零方法：5+10 和 5*3，优先找大额的
                ten -= 1
                five -= 1
            else:
                five -= 3

            # elif bill = 20:
            #     # 收一个20美元，两种找零方法：5+10 和 5*3，优先找大额的
            #     if ten > 0:
            #         ten -= 1
            #         five -= 1
            #     else:
            #         five -= 3
```

```
            if five < 0:        # error: 缩进，ten < 0 其实已经包含在上面了，所以不用
写
                # 如果5和10的个数小于0，则False
                return False

        return True
```

一行写：

```
    def lemonadeChange(self, bills):
        five = ten = 0
        for i in bills:
            if i == 5: five += 1
            elif i == 10: five, ten = five - 1, ten + 1
            elif ten > 0: five, ten = five - 1, ten - 1
            else: five -= 3
            if five < 0: return False
        return True
```

Java

```
class Solution {
    public boolean lemonadeChange(int[] bills) {
        // 2ms, beats 99.96%
        int five = 0, ten = 0;
        for (int bill : bills) {
            if (bill == 5) {
                five++;
            }
            else if (bill == 10) {
                five--;
                ten++;
            } else if (ten > 0) {     // !!!
                    five--;
                    ten--;
            } else {
                    five -= 3;
            }
            // 因为ten < 0 找five，ten < 0 其实已经包含在上面了，所以不用谢
            if (five < 0) {           // ||或，&&与
                return false;
            }
        }

        return true;
    }
}
```

一行写：

```
    public boolean lemonadeChange(int[] bills) {
        int five = 0, ten = 0;
        for (int i : bills) {
            if (i == 5) five++;
            else if (i == 10) {five--; ten++;}
            else if (ten > 0) {ten--; five--;}
            else five -= 3;
            if (five < 0) return false;
        }
        return true;
    }
```

C++

```
    int lemonadeChange(vector<int> bills) {
        int five = 0, ten = 0;
        for (int i : bills) {
            if (i == 5) five++;
            else if (i == 10) five--, ten++;
            else if (ten > 0) ten--, five--;
            else five -= 3;
            if (five < 0) return false;
        }
        return true;
    }
```

121. 买卖股票的最佳时机

难度简单1050

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

贪心分析： 一遍循环

我们只需要遍历价格数组一遍，**记录历史最低点**，然后在每一天考虑这么一个问题：如果我是在历史最低点买进的，那么我今天卖出能赚多少钱？当考虑完所有天数之时，我们就得到了最好的答案。
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/solution/121-mai-mai-gu-piao-de-zui-jia-shi-ji-by-leetcode-/

代码

Python *******

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 审题
        # 算法
        # 1 两轮遍历，外循环是买价格，内循环是卖价格，差 = 卖 -  买，找差的最大值
        # time：O(N*2)
        # 2 优化：一轮遍历，记录前面买的最低值，当天价格作为卖出值，找差的最大值
        # time: O(N)
        # space: O(1)

        # special condition
        if len(prices) < 2: return 0
        # if not prices: return 0       # it's ok

        max_profit = 0
        # since must buy first, so I think it's better than 1e9??
        min_buy_price = prices[0]

        # since must buy first, so from the 2nd one
        for price in prices[1:]:
            min_buy_price = min(min_buy_price, price)
            # if price < min_buy_price:
            #     min_buy_price = price
            max_profit = max(max_profit, (price - min_buy_price))
        return max_profit
```

暴力分析：两轮循环

两轮遍历，外循环是买价格，内循环是卖价格，差 = 卖 - 买，找差的最大值

代码

Python

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 审题
        # 算法
        # 1 两轮遍历，外循环是买价格，内循环是卖价格，差 = 卖 -  买，找差的最大值
        # time：O(N*2)
        # timeout, too slow

        max_profit = 0

        for buy_idx in range(len(prices) - 1):     # -1
            for sell_idx in range(buy_idx + 1, len(prices)):
```

```
                max_profit = max(max_profit, (prices[sell_idx] -
prices[buy_idx]))
                # profit = prices[sell_idx] - prices[buy_idx]
                # if profit > max_profit:
                #     max_profit = profit

        return max_profit
```

Java

```
class Solution {
    public int maxProfit(int[] prices) {
        # time: 343ms, beats 7.36%, slow
        int max_profit = 0;

        for (int buy_idx = 0; buy_idx < prices.length - 1; buy_idx++) {
            for (int sell_idx = buy_idx + 1; sell_idx < prices.length;
sell_idx++) {
                int profit = prices[sell_idx] - prices[buy_idx];
                if (profit > max_profit) {
                    max_profit = profit;
                }
            }
        }
        return max_profit;
    }
}
```

122. 买卖股票的最佳时机 II

难度简单753

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能
获得利润 = 5-1 = 4 。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交
易所能获得利润 = 6-3 = 3 。
```

示例 2:

```
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能
获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

提示:

- `1 <= prices.length <= 3 * 10 ^ 4`
- `0 <= prices[i] <= 10 ^ 4`

贪心分析

股票买卖策略:

单独交易日： 设今天价格 p1, 明天价格 p2, 则今天买入、明天卖出可赚取金额 p2 - p1（负值代表亏损）。
连续上涨交易日： 设此上涨交易日股票价格分别为 p1,p2...pn, 则第一天买最后一天卖收益最大，即 pn - p1;
等价于每天都买卖，即 pn - p1=(p2 - p1)+(p3 - p2)+...+(pn - pn-1)

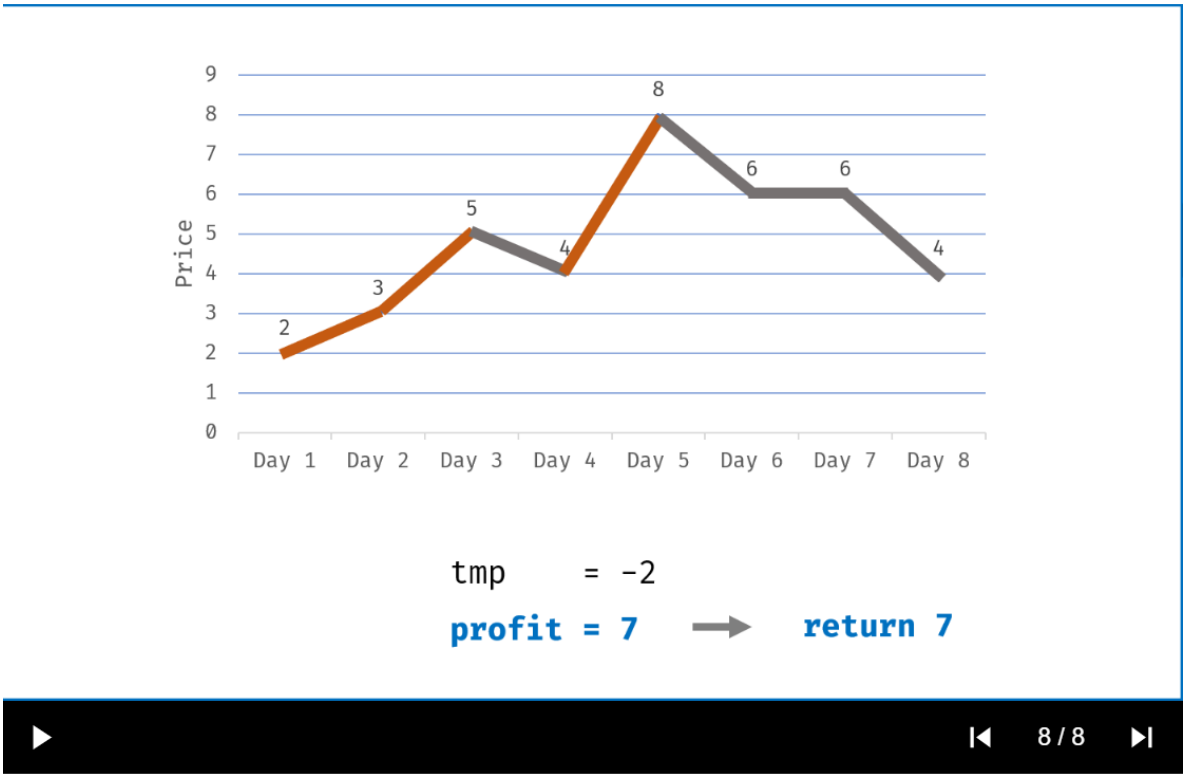
连续下降交易日： 则不买卖收益最大，即不会亏钱。

算法流程:

遍历整个股票交易日价格列表 price，策略是所有上涨交易日都买卖（赚到所有利润），所有下降交易日都不买卖（永不亏钱）。
设 tmp 为第 i-1 日买入与第 i 日卖出赚取的利润，即 tmp = prices[i] - prices[i - 1]；
当该天利润为正 tmp > 0，则将利润加入总利润 profit；当利润为 00 或为负，则直接跳过；
遍历完成后，返回总利润 profit。

复杂度分析:

时间复杂度 O(N)O(N)： 只需遍历一次price；
空间复杂度 O(1)O(1)： 变量使用常数额外空间。
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/solution/best-time-to-buy-and-sell-stock-ii-zhuan-hua-fa-ji/



You can do it on a day-to-day basis. If buying on day 1 and selling on day 2 is profitable, do it. If buying on day 2 and selling on day 3 is profitable, do it. And so on. Yes, you can do **both** day1-to-day2 **and** day2-to-day3, even though there are multiple transactions on day 2. Either think of it as selling first and then buying later on that day, or think of it as **keeping** instead of selling+buying.

代码

python ********

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # time: 76ms, beats 69%
        if len(prices) < 2:          # error: deal with special !!!
            return 0
        res = 0
        for i in range(len(prices) - 1):
            profit = prices[i + 1] - prices[i]
            if profit > 0:
                res += profit
        return res
```

一行总结tips：比较大小，就用max，min，sum；生成式**

one-line python *******

```
class Solution(object):
    def maxProfit(self, prices):
        return sum(max(prices[i + 1] - prices[i], 0) for i in range(len(prices)
- 1))
        # error: no [], [max(,) for i in range(...)]
```

Python by StefanPochmann

Python's `zip` is also quite nice, and you can give it lists of different sizes, which none of the similar solutions I've seen from others exploited.

```
def maxProfit(self, prices):
    return sum(b-a for a,b in zip(prices,prices[1:])if b>a)
```

Or:

```
def maxProfit(self, prices):
    return sum(max(b-a,0)for a,b in zip(prices,prices[1:]))
```

455. 分发饼干

https://leetcode-cn.com/problems/assign-cookies/submissions/

难度简单180

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i ，都有一个胃口值 gi ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 sj 。如果 sj >= gi ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

注意：

你可以假设胃口值为正。
一个小朋友最多只能拥有一块饼干。

示例 1:

```
输入: [1,2,3], [1,1]

输出: 1

解释:
你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。
虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。
所以你应该输出1。
```

示例 2:

```
输入: [1,2], [1,2,3]

输出: 2

解释:
你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。
你拥有的饼干数量和尺寸都足以让所有孩子满足。
所以你应该输出2.
```

贪心分析：

1. 给一个孩子的饼干应当尽量小并且又能满足该孩子，这样大饼干才能拿来给满足度比较大的孩子。

2. 因为满足度最小的孩子最容易得到满足，所以先满足满足度最小的孩子。
3. 一个孩子只能用一个盘子里的饼干满足，不够就整个盘子都不能要了。

代码

python ******

```
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        """
        审题：return 满足孩子的最大值，从好满足的/小的 开始
        time：192ms, beats 97.58%
        """
        # g和s由小到大排序
        g.sort()
        s.sort()
        # 定义初始下标为0
        g_child = 0
        s_cookie = 0

        while s_cookie < len(s) and g_child < len(g):
            # 如果第2盘饼干，可以给第2个小朋友，则各+1
            # 如果第2盘饼干，无法给第2个小朋友，就试第3盘饼干（饼干+1），因为第3盘饼干数量多于第2盘
            # 小朋友不变
            if s[s_cookie] >= g[g_child]:                # error：=就够吃了
                g_child += 1                             #
            s_cookie += 1                                #

        return g_child
```

55. 跳跃游戏

难度中等719

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1，然后再从位置 1 跳 3 步到达最后一个位置。
```

示例 2:

```
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。
```

贪心分析 + 正序

正序遍历每个值，设置可以到达的最远位置为0

更新可以到达的最远位置，

如果大于等于最后元素位置，就return true；

如果小于当前位置，return false

- 方法 1：可以到达最远位置

代码

Python ******

贪心正向查找，每次达到最远的位置，则总步数最小

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        # time: O(N)
        # space: O(1)
        max_reach, end, count = 0, 0, 0
        for i in range(len(nums) - 1):
            if i <= max_reach:
                max_reach = max(max_reach, nums[i] + i)
                if i == end:
                    end = max_reach
                    count += 1

        return count
```

优化代码：

```
pyclass Solution:
    def canJump(self, nums: List[int]) -> bool:
        # time: 60ms, beats 40%
        # time:
        max_reach, n = 0, len(nums)

        for i, x in enumerate(nums):
            if max_reach < i: return False
            if max_reach >= n - 1: return True
            max_reach = max(max_reach, i + x)
```

倒序分析

代码

python

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        # 从数组倒数第二个元素，倒序遍历到第一个元素
        # 需要到达终点的步数需求need default = 1
        # 如果元素值 < need，则此位置无法到终点，
        # 考虑前一个元素，need + 1; continue
        # 如果元素值 >= need，则此位置可以到终点，
        # 此位置即为新终点，need 重置为1；继续向前遍历
        # time: O(N)
        # space: O(1)

        # special
        if len(nums) < 2: return True

        result, need = True, 1

        for i in range(len(nums) - 2, -1, -1):
            if nums[i] < need:
                # 如果元素值 < need，则此位置无法到终点，
                # 考虑前一个元素，need + 1; continue
                need += 1
                result = False
                continue
            else:
                # 如果元素值 >= need，则此位置可以到终点，
                # 此位置即为新终点，need 重置为1；继续向前遍历
                need = 1
                result = True
                continue

        return result
```

45. 跳跃游戏 II

难度困难610

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

```
输入: [2,3,1,1,4]
输出: 2
解释: 跳到最后一个位置的最小跳跃数是 2。
     从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
```

贪心分析

代码

Python******

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        # 贪心，每次都到能达到的最大位置，则整体最优
        # time: O(N)
        # space: O(1)
        # time: 44ms, beats 97.74%
        max_reach, end, count = 0, 0, 0
        for i in range(len(nums) - 1):
            if i <= max_reach:
                max_reach = max(max_reach, nums[i] + i)
                if i == end:             # ???
                    end = max_reach
                    count += 1

        return count
```

https://leetcode-cn.com/problems/jump-game-ii/solution/tiao-yue-you-xi-ii-by-leetcode-solution/