213. House Robber II
We will use the same way to check nums[0:] and nums[1:] to get the maximum, so that we don't need to care about whether there are conflicts between the first element and the last one. TC is O(n), SC is O(1)

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        if not nums:
          return 0

        length = len(nums)
        if length == 1:
          return nums[0]

        return max(self.helper(nums[0:length-1]), self.helper(nums[1:length]))

    def helper(self, nums):
        prev1, prev2 = 0, nums[0]

        for i in range(1, len(nums)):
          cur_max = max(prev1 + nums[i], prev2)
          prev1, prev2 = prev2, cur_max
        return max(prev1, prev2)
```

309. Best Time to Buy and Sell Stock with Cooldown
We have three states: s1, s2, s3 and their relationships: s1->rest->s1, s1-> buy-> s2, s2->rest->s2, s2->sell->s3, s3->rest->s1, so

```
        S1[i] = max(s1[i - 1], s3[i - 1])
        S2[i] = max(s1[i - 1] - prices[i], s2[i - 1]))
        S3[i] = s2[i - 1] + prices[i]
```

TC is O(n), SC is O(n)

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
      if not prices:
        return 0
      s1, s2, s3 = [0], [-prices[0]], [0]
      for i in range(1, len(prices)):
        s1.append(max(s1[i - 1], s3[i - 1]))
        s2.append(max(s1[i - 1] - prices[i], s2[i - 1]))
        s3.append(s2[i - 1] + prices[i])
      return max(s1[-1], s3[-1])
```

740. Delete and Earn

We will count all elements in nums and then it's similar to find the maximum sum of elements that cannot be continuous. TC is O(n), SC is O(n).

```
from collections import Counter
class Solution:
    def deleteAndEarn(self, nums: List[int]) -> int:
        if not nums:
            return 0
        points = Counter(nums)
        min_num, max_num = min(nums), max(nums)
        prev, curr = 0, 0
        for i in range(min_num, max_num + 1):
            prev, curr = curr, max(curr, points[i] * i + prev)
        return curr
```

790. Domino and Tromino Tiling

We will try to find regulation among the first several results and found that nums[i] = 2 * nums[n - 1] + nums[n - 3], TC is O(n), SC is O(n)

```
class Solution:
    def numTilings(self, N: int) -> int:
        nums = [1, 2, 5]
        if N < 4:
            return nums[N - 1]
        for i in range(3, N):
            nums.append((2 * nums[i - 1] + nums[i - 3]) % (10 ** 9 + 7))
        return nums[-1]
```

801. Minimum Swaps To Make Sequences Increasing

We will assign swap and not_swap to record minimum time we need to swap to make two arrays both increasing. There are three situations we need to talk about:

1. A[i - 1] >= B[i] or B[i - 1] >= A[i]:
   Swap[i] = Swap[i - 1] + 1
   Not_swap[i] = Not_swap[i - 1]
2. B[i - 1] >= B[i] or A[i - 1] >= A[i]:
   Swap[i] = Not_swap[i - 1] + 1
   Not_swap[i] = Swap[i - 1]
3. Temp = min(Not_swap[i - 1], Swap[i - 1])
   Swap[i] = Temp + 1
   Not_swap[i] = Temp

TC is O(n), SC is O(1)

```
class Solution:
```

```
def minSwap(self, A: List[int], B: List[int]) -> int:
    swap = 1
    not_swap = 0
    for i in range(1, len(A)):
      if A[i - 1] >= B[i] or B[i - 1] >= A[i]:
        swap += 1
      elif A[i - 1] >= A[i] or B[i - 1] >= B[i]:
        swap, not_swap = not_swap + 1 , swap
      else:
        not_swap = min(swap, not_swap)
        swap = not_swap + 1
    return min(not_swap, swap)
```