# Array

## 442. Find All Duplicates in an Array

Given an array of integers, 1≤a[i]≤ *n* (*n* = size of array), some elements appear **twice** and others appear **once**. Find all the elements that appear **twice** in this array. Could you do it without extra space and in O(*n*) runtime?

1. **def** findDuplicates(self, nums):
2. """
3. :type nums: List[int]
4. :rtype: List[int]
5. """
6. dupl = []
7. numdir={}
8. **for** v **in** nums:
9. **if** v **in** numdir:
10. dupl.append(v)
11. **else**:
12. numdir[v] = 1
13. **return** dupl

上述代码使用了额外的空间。

1. **def** findDuplicates(self, nums):
2. """
3. :type nums: List[int]
4. :rtype: List[int]
5. """
6. dupl = []
7. **for** x **in** nums:
8. **if** nums[abs(x)-1] < 0:
9. dupl.append(abs(x))
10. **else**:
11. nums[abs(x)-1]\*=-1
12. **return** dupl

上面的代码使用nums本身进行原值排序，把x的信息映射到nums[abs(x)-1]上，用数字的符号标记出现的次数。并且使用abs()进行数值还原。

## 495. Teemo Attacking

In LOL world, there is a hero called Teemo and his attacking can make his enemy Ashe be in poisoned condition. Now, given the Teemo's attacking ascending time series towards Ashe and the poisoning time duration per Teemo's attacking, you need to output the total time that Ashe is in poisoned condition.

You may assume that Teemo attacks at the very beginning of a specific time point, and makes Ashe be in poisoned condition immediately.

思路：判断连续两次攻击的间隔和麻痹时间的大小，取较短的时间为有效的麻痹时间。注意如果攻击序列为空时的判断。Python中range()下表有误，不会报错，只会返回一个空列表。

1. **def** findPoisonedDuration(self, timeSeries, duration):
2. """
3. :type timeSeries: List[int]
4. :type duration: int
5. :rtype: int
6. """
7. pt = 0
8. lent = len(timeSeries)
9. **for** i **in** list(range(1,lent)):
10. pt=pt+min(timeSeries[i]-timeSeries[i-1],duration)
12. **if** lent>0:
13. pt=pt+duration
14. **return** pt

但是下面的方法与上面的方法有着相同的含义，但是代码极为简洁，优美！less is more!

Just calculate sum(min(duration, next attack time - this attack time))

1. **class** Solution:
2. **def** findPoisonedDuration(self, timeSeries, duration):
3. """
4. :type timeSeries: List[int]
5. :type duration: int
6. :rtype: int
7. """
8. **return** sum(min(duration, b - a) **for** a, b **in** zip(timeSeries, timeSeries[1:] + [10e9]))

## 59. Spiral Matrix II

Given an integer *n*, generate a square matrix filled with elements from 1 to *n*2 in spiral order.

这题我在控制转向时，检测时没有控制好导致数组越界。并且逻辑混乱

1. **def** generateMatrix(n):
2. """
3. :type n: int
4. :rtype: List[List[int]]
5. """
6. matrix = [[0] \* n **for** i **in** range(n)]  #Python中生成多维数组的方法
7. v = 1
8. i, j, di, dj = 0, 0, 0, 1
9. **while** v <= n \* n:
10. matrix[i][j] = v
11. **if** matrix[(i + di) % n][(j + dj) % n]:#检测越界的方式很值得借鉴
12. di, dj = dj, -di #改变移动方向
13. i = i + di
14. j = j + dj
15. v = v + 1
16. **return** matrix

从上面代码可以学习到，转向的方式。定义移动方向并且方向变换（很六）。利用模运算检测是否越界，在Python中-1%5 = 4，

对于任何同号的两个整数，其取余结果没有争议，所有语言的运算原则都是使商尽可能小。

对于异号的两个整数，C++/Java语言的原则是使商尽可能大，很多新型语言Python等原则是使商尽可能小。

## [121. Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/" \l "/description)

Say you have an array for which the *i*th element is the price of a given stock on day *i*. If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

**Example 1:**

Input: [7, 1, 5, 3, 6, 4]

Output: 5

max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)

**Example 2:**

Input: [7, 6, 4, 3, 1]

Output: 0

In this case, no transaction is done, i.e. max profit = 0.

解法一(分治算法):

我们需要寻找A[low,…,high]中最大的子数组。分治算法要求我们将将数组分成两个规模尽量相等的子数组，A[low,…,mid]和A[mid+1,…,high]两部分。最大子数组序列A[i,…,j]必然是三种情况之一：

* 完全位于子数组A[low,…,mid]中,此时low<= i<=j<=mid
* 完全位于子数组A[mid+1,…,high]中,此时mid+1<= i<=j<=high
* 跨越了中点mid，此时low<= i<=mid<j<=high

对于前两种情况，可以使用递归解决。第三种情况并不是原问题规模更小的子问题了!因为它加入了限制条件。子数组必须跨越中点。因此我们只需要找到形如A[i,…mid]和A[mid,…,j]中的最大子数组，然后讲他们合并即可。

1. **def** maxmidsum(self,low,mid,high,pricewave):
2. leftsum = 0
3. rightsum = 0
4. sum = 0
5. **for** i **in** list(range(mid,low-1,-1)):
6. sum+=pricewave[i]
7. **if** sum > leftsum:
8. leftsum = sum
9. sum = 0
10. **for** j **in** list(range(mid+1,high+1)):
11. sum+=pricewave[j]
12. **if** sum > rightsum:
13. rightsum = sum
14. **return** leftsum+rightsum

然后我们就可以通过调用递归的方式来进行调用了。

1. **def** findmaxsum(self,low,high,pricewave):
2. mid = (low+high)//2
3. **if** low == high:
4. **return** pricewave[low]
5. **else**:
6. leftsum = self.findmaxsum(low,mid,pricewave)  #左边最大子数组和
7. rightsum = self.findmaxsum(mid+1,high,pricewave)  #右边最大子数组和
8. midsum = self.maxmidsum(low,mid,high,pricewave)  #跨中点最大子数组
9. **return** max(max(leftsum,rightsum),midsum)

在主方法中进行调用。注意在类中定义方法时需要定义self这个参数

1. **def** maxProfit(self, prices):
2. """
3. :type prices: List[int]
4. :rtype: int
5. """
6. **if** len(prices)<=1:
7. **return** 0
8. low = 0
9. pricewave = [a-b **for** a,b **in** zip(prices[1:]+[0],prices)][:-1]
10. high = len(pricewave)-1
11. **return** max(self.findmaxsum(low,high,pricewave),0)

解法二(dp):

假设最大和子序列的起点为A[i],那么A[i]一定是正数，否则可以通过去掉A[i]获得一个更优的子序列。

同理，任何一个负的子序列都不可能是最优子序列的前缀。

假设MAXSUM[i]表示前i个最大子序列的和。

MAXSUM[i]=max{ MAXSUM[i-1]+A[i],A[i]}

1. **class** Solution:
2. **def** maxProfit(self, prices):
3. """
4. :type prices: List[int]
5. :rtype: int
6. """
7. currentsum,maxsum = 0,0
8. **for** i **in** list(range(1,len(prices))):
9. currentsum = max(0,currentsum+prices[i]-prices[i-1])
10. maxsum = max(maxsum,currentsum)
11. **return** maxsum

第三种方法：

最大和子序列的起点一定是最小值在前，最大值在后面，找到这样的最大山峰，和最小的最小的山谷。We need to find the largest peak following the smallest valley.

1. **class** Solution:
2. **def** maxProfit(self, prices):
3. """
4. :type prices: List[int]
5. :rtype: int
6. """
7. **if** len(prices)==0:
8. **return** 0
9. minprice = prices[0]
10. maxprofit = 0
11. **for** p **in** prices:
12. **if** p < minprice:
13. minprice=p
14. **else**:
15. maxprofit = max(maxprofit,p-minprice)
16. **return** maxprofit

注意特殊的输入数据，比如输入为空的情况。

## [122. Best Time to Buy and Sell Stock II](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/#/description)

Say you have an array for which the *i*th element is the price of a given stock on day *i*.Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

思路一：（动态规划）

利用S0表示第i天后不持有股票的收益，S1表示第i天后持有股票带来的收益。第i天后不持有股票有两种可能，第i-1天不持有股票第i天把股票卖出了。第i天后持有股票有两种可能，第i-1天持有股票第i天把股票买出了。

因此可以得到如下的递推公式

S0 = max(S0,S1+P)

S1 = max(S1,S0-P)

1. **class** Solution:
2. def maxProfit(self, prices):
3. """
4. :type prices: List[**int**]
5. :rtype: **int**
6. """
7. **if** len(prices)<2:
8. **return** 0
9. s0,s1 = 0,-prices[0] #初始化
10. **for** p in prices[1:]:
11. s0 = max(s0,s1+p)
12. s1 = max(s1,s0-p)
13. **return** s0

思路二：（找到所有的波谷和波峰）



表示紧跟在波谷i后面的波峰i。也就是说所有的增长差。

1. **class** Solution:
2. def maxProfit(self, prices):
3. """
4. :type prices: List[**int**]
5. :rtype: **int**
6. """
7. **if** len(prices)<2:
8. **return** 0
9. totalprofit = 0
10. **for** i in list(range(1,len(prices))):
11. **if** prices[i]>prices[i-1]:
12. totalprofit+=prices[i]-prices[i-1]
13. **return** totalprofit

## [123. Best Time to Buy and Sell Stock III](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/#/description)

Say you have an array for which the *i*th element is the price of a given stock on day *i*. Design an algorithm to find the maximum profit. You may complete atmost *two* transactions.

**Note:**  
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

思路均是 动态规划

方法一，专用方法只针对这一题。

buy1[i]表示在第i天买入第一支股票的收益，第一天我们需要借钱买股票（buy[i]是负数）。因此，我们希望他越大越好。

sell1[i]表示在第i天卖出第一支股票的收益，我们希望卖出的收益越大越好。prices[i ]+buy1[i-1] 是卖出后的收益。

Buy2[i]表示在第i天买入第二支股票的收益，此时我们有本金sell1[i-1]。因此，买入后我们还有sell1[i-1]- prices[i ]

Sell2[i]表示在第i天卖出第二支股票的收益，我们希望卖出的收益越大越好。prices[i ]+buy2[i-1] 是卖出后的收益。

数学表达式如下：

1. ***buy1[i] = max( - prices[i], buy1[i - 1])***
2. ***sell1[i] = max(buy1[i - 1] + price[i], sell1[i - 1])***
3. ***buy2[i] = max( sell1[i -1] - prices[i], buy2[i - 1])***
4. ***sell2[i] = max(buy2[i - 1] + price[i], sell2[i - 1])***

但是，具体编程不需要使用数组来存储中间结果，只需要使用两个变量即可。

1. **class** Solution:
2. **def** maxProfit(self, prices):
3. """
4. :type prices: List[int]
5. :rtype: int
6. """
7. K=2
8. length = len(prices)
9. **if** length<2:
10. **return** 0
11. buy1,buy2=-10000000000000,-10000000000000
12. sell1,sell2 = 0,0
13. **for** p **in** prices:
14. buy1 = max(buy1, -p)
15. sell1 = max(sell1, buy1+p)
16. buy2 = max(buy2, sell1-p)
17. sell2 = max(sell2,buy2+p)
18. **return** sell2

## [188. Best Time to Buy and Sell Stock IV](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/#/description)

对上述方法进行扩展，可以处理任意交易k的最大值。

假设profit[k][i]中存储前i天交易k次所获得的最大利润，那么

profit[k][i]=max{ profit[k][i-1], max(profit[k-1][j]+prices[i]-prices[j], 0<=j<i ） }

= max{ profit[k][i-1], prices[i]+max(profit[k-1][j]-prices[j], 0<=j<i） }

注意内部最大值j的求法。不能使用循环来求解最大值，否则将会出现超时。下面的算法将会出现超时的情况。这里也不需要使用第三重循环来求解这个问题，这里只需要使用。

1. **class** Solution:
2. **def** maxProfit(self, prices):
3. """
4. :type prices: List[int]
5. :rtype: int
6. """
7. K=2
8. length = len(prices)
9. **if** length<2:
10. **return** 0
11. profit = [[0]\*(length) **for** i **in** range(K+1)]
12. **for** k **in** range(1,K+1):
13. **for** i **in** list(range(1,length)):
14. **for** j **in** range(i):
15. profit[k][i] = max(profit[k][i-1],profit[k-1][j]+prices[i]-prices[j])
16. **return** profit

这里profit[k-1][i-1]-prices[i-1]的含义是在第i-1天买入股票，那么在第i天就可以卖出股票。这里充分利用了profit[k-1][i-1]的含义（第i-1天完成k-1笔交易的收益，利用i的变化完成计数）

1. **class** Solution:
2. **def** maxProfit(self, prices):
3. K=2
4. length = len(prices)
5. **if** length<2:
6. **return** 0
7. profit = [[0]\*(length) **for** i **in** range(K+1)]
8. **for** k **in** range(1,K+1):
9. maxtmp = profit[k][0]-prices[0]
10. **for** i **in** list(range(1,length)):
11. maxtmp = max(maxtmp,profit[k-1][i-1]-prices[i-1])
12. profit[k][i] = max(profit[k][i-1],prices[i]+maxtmp)
13. **return** profit[k][length-1]

考虑特殊的数据，采用如下的方法。但是这样的解法很不优雅。

1. **class** Solution:
2. **def** maxProfit(self, k, prices):
3. """
4. :type k: int
5. :type prices: List[int]
6. :rtype: int
7. """
8. length = len(prices)
9. **if** length<2 **or** k<1:
10. **return** 0
12. **if** k==1:#如果k=1采用最大波峰波谷法进行解决
13. minprice,maxprofit = prices[0],0
14. **for** p **in** prices[1:]:
15. **if** p < minprice:
16. minprice = p
17. **else**:
18. maxprofit = max(maxprofit,p-minprice)
19. **return** maxprofit
20. **elif** k > len(prices):#如果k大于交易天数，交易次数不限。那么采用求所有价格差的方法
21. totalprofit = 0
22. **for** i **in** list(range(1,len(prices))):
23. **if** prices[i] > prices[i-1]:
24. totalprofit+=(prices[i]-prices[i-1])
25. **return** totalprofit
26. **else**:#真正的方法
27. profit = [[0]\*(length) **for** i **in** range(k+1)]
28. **for** i **in** range(1,k+1):
29. maxtmp = profit[i][0]-prices[0]
30. **for** j **in** list(range(1,length)):
31. profit[i][j] = max(profit[i][j-1],prices[j]+maxtmp)
32. maxtmp = max(maxtmp,profit[i-1][j]-prices[j])
33. **return** profit[k][length-1]

## [309. Best Time to Buy and Sell Stock with Cooldown](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/#/description)

Say you have an array for which the *i*th element is the price of a given stock on day *i*.Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

* You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
* After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

很自然的想到使用每天的交易动作将问题划分开

Buy[i]表示前i天以买入股票为最后动作的序列带来的最大收益。

Sell[i]表示前i天以卖出股票带为最后动为最后序列带来的最大收益。

Rest[i]表示前i天以没有交易为最后序列带来的最大收益。

Buy[i] = max(Buy[i-1], Rest[i-1]-p)

Sell[i] = max(Sell[i-1], Buy[i-1]+p)

Rest[i] = max(Buy[i-1],Sell[i-1],Rest[i-1])

由于Sell[i]>= Rest[i-1]>= Buy[i-1]所以Rest[i-1]= Sell[i-1]

Buy[i] = max(Buy[i-1], Sell [i-2]-p)

Sell[i] = max(Sell[i-1], Buy[i-1]+p)

1. **class** Solution:
2. **def** maxProfit(self, prices):
3. **if** len(prices) < 2:
4. **return** 0
5. sell, buy, prev\_sell, prev\_buy = 0, -prices[0], 0, 0
6. **for** price **in** prices:
7. prev\_buy = buy
8. buy = max(prev\_sell - price, prev\_buy)
9. prev\_sell = sell
10. sell = max(prev\_buy + price, prev\_sell)
11. **return** sell

## [714. Best Time to Buy and Sell Stock with Transaction Fee](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/description/)

Your are given an array of integers prices, for which the i-th element is the price of a given stock on day i; and a non-negative integer fee representing a transaction fee. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. You may not buy more than 1 share of a stock at a time (ie. you must sell the stock share before you buy again.). Return the maximum profit you can make.

相似的思路将函数的状态划分为

Buy[i]表示前i天以买入股票为最后动作的序列带来的最大收益。

Sell[i]表示前i天以卖出股票带为最后动为最后序列带来的最大收益。

注意一次完整的交易包含买和卖两次过程，因此可以选择在买入时缴纳手续费或者在卖出时缴纳手续费。本文中以股票卖出为一次交易的结束，此时缴纳手续费。

Buy[i] = max(Buy[i-1], Sell [i-1]-p)

Sell[i] = max(Sell[i-1], Buy[i-1]+p-free)

1. **class** Solution:
2. **def** maxProfit(self, prices, fee):
3. """
4. :type prices: List[int]
5. :type fee: int
6. :rtype: int
7. """
8. sell, buy = 0, -500010
9. **for** i **in** range(0, len(prices)):
10. sell = max(sell, buy + prices[i] - fee)
11. buy = max(buy, sell - prices[i])
12. **return** sell

## [summary](https://www.baidu.com/link?url=weVkL2tHIEAjUo2ES0G0zBHMiVorKvfb8IyA95KS4CaDFktbny2_24K-cQVmgQfBbUCNtyBxh0DRMfqV9pxWsXMOcvN7M_fUTMUv1hvjpEW&wd=&eqid=d33cbaae00009c14000000035aa38e60)：Most consistent ways of dealing with the series of stock problems

一下是leetcode中关于股票问题的合集

1. [121. Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/#/description)
2. [122. Best Time to Buy and Sell Stock II](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/#/description)
3. [123. Best Time to Buy and Sell Stock III](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/#/description)
4. [188. Best Time to Buy and Sell Stock IV](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/#/description)
5. [309. Best Time to Buy and Sell Stock with Cooldown](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/#/description)
6. [714. Best Time to Buy and Sell Stock with Transaction Fee](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/description/)

### 1.通用情况

假设prices 表示股票价格序列，序列长度为n,k表示最大交易次数。由于卖出股票的前提是手中有股票，因此采用如下的子状态T[i][k][0] 和 T[i][k][1]。T[i][k][0] 表示第i-th天结束时，最多交易k次，最后手里剩下0个股票的最大收益。T[i][k][0] 表示第i-th天结束时，最多交易k次，最后手里剩下1个股票的最大收益。那么状态转移方程为如下：

1. Base cases:  
   T[-1][k][0] = 0, T[-1][k][1] = -Infinity  
   T[i][0][0] = 0, T[i][0][1] = -Infinity
2. Recurrence relation:  
   T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])  
   T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])

T[-1][k][0] = 0(i=-1表示no stock)和T[i][0][0] = 0(k=0表示没有交易)。 T[-1][k][1] = -Infinity和T[i][0][1] = -Infinity 表示强调不可能拥有股票在没有股票或者没有交易的情况下。

由于每天的状态可以分为三种。Buy(买入),Sell(卖出)和Rest(不交易)。

T[i][k][0](第i天过后手中没有股票) = max(T[i-1][k][0](第i-1天结束时手里就没有股票，第i天不交易 rest), T[i-1][k][1] + prices[i](第i-1天结束时手里就有一张股票, 第i天卖出sell))  
T[i][k][1] (第i天过后手中有一张股票) = max(T[i-1][k][1](第i-1天结束时手里有股票，第i天不交易 rest), T[i-1][k-1][0] - prices[i] (第i-1天结束时手中没有股票, 第i天买入股票 buy))

### 2.带有限制条件的特殊情况

**情况1: k=1**

[121. Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/#/description)

T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])  
T[i][1][1] = max(T[i-1][1][1], T[i-1][0][0] - prices[i])

= max(T[i-1][1][1], -prices[i])

由于第i天的情况只依赖第i-1天，因此可以进行优化。

**情况2: k=不限次数,**

[122. Best Time to Buy and Sell Stock II](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/#/description)

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-1][k-1][0] - prices[i])

**情况3: k=2**

[123. Best Time to Buy and Sell Stock III](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/#/description)

T[i][2][0] = max(T[i-1][2][0], T[i-1][2][1] + prices[i])

T[i][2][1] = max(T[i-1][2][1], T[i-1][1][0] - prices[i])

T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])

T[i][1][1] = max(T[i-1][1][1], -prices[i])

**情况4: k=指定次数**

由于一天只能买入或者卖出，因此多余的交易次数并不能继续带来利润的增加，因此当交易次数k大于股票序列长度的一般后就不能带来收益的增加了。此时现在交易次数就没有意义了，可以扩展成情况二。

**情况5: k=不限次数，带有冷却时间(2)的**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])  
T[i][k][1] = max(T[i-1][k][1], T[i-2][k][0](不用T[i-1][k][0]表示冷却时间) - prices[i])

**情况6: k=不限次数，带有交易费用**

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])  
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i] - fee)

or

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i] - fee)  
T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])

一次交易分为买和卖两次过程，因此可以在买或者的时候缴纳交易费用。

## 717. 1-bit and 2-bit Characters

We have two special characters. The first character can be represented by one bit 0. The second character can be represented by two bits (10 or 11).Now given a string represented by several bits. Return whether the last character must be a one-bit character or not. The given string will always end with a zero.

解法一:

从头开始遍历整个数组，如果第i个bit是0,表示则光标向后移动一个位置，则光标向后移动一个位置指向下一个字符位置。如果第i个bit是1,表示则光标向后移动两个位置，指向下一个字符位置。最后判断光标是否指最后一个字符位置即可。

1. **class** Solution:
2. **def** isOneBitCharacter(self, bits):
3. """
4. :type bits: List[int]
5. :rtype: bool
6. """
7. **if** len(bits)==1:
8. **return** True
9. i = 0
10. **while** i < (len(bits)-1):
11. **if** bits[i]==0:
12. i=i+1
13. **else**:
14. i=i+2
15. **return** i == (len(bits)-1)

解法二:

我们从后向前遍历整个数组，在我们遇到下一个零之前，编码1的个数应该是偶数个。

1. **class** Solution:
2. **def** isOneBitCharacter(self, bits):
3. """
4. :type bits: List[int]
5. :rtype: bool
6. """
7. **if** len(bits)==1:
8. **return** True
9. ones = 0
10. **for** i **in** list(range(len(bits)-2,-1,-1)):
11. **if** bits[i]==1:
12. ones+=1
13. **else**:
14. **break**
15. **return** ones%2==0

## [1. Two Sum](https://leetcode.com/problems/two-sum/description/)

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

Given nums = [2, 7, 11, 15], target = 9,

Because nums[**0**] + nums[**1**] = 2 + 7 = 9,

return [**0**, **1**].

思路：

利用hash表进行处理。将nums[i]作为key值,数组下标作为i作为value值构建hash表。利用hash表进行查找。

1. **class** Solution:
2. **def** twoSum(self, nums, target):
3. """
4. :type nums: List[int]
5. :type target: int
6. :rtype: List[int]
7. """
8. numdirct = {}
9. **for** i **in** range(len(nums)):
10. numdirct[nums[i]] = i
12. **for** i **in** range(len(nums)):
13. **if** target-nums[i] **in** numdirct:
14. **if** i != numdirct[target-nums[i]]:
15. **return**[i,numdirct[target-nums[i]]]

## 15. [3Sum](https://leetcode.com/problems/3sum/description/)

Given an array *S* of *n* integers, are there elements *a*, *b*, *c* in *S* such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.**Note:** The solution set must not contain duplicate triplets.

For example, given array S = [-1, 0, 1, 2, -1, -4],

A solution set is:

[

[-1, 0, 1],

[-1, -1, 2]

]

思路：先对数组排序，先利用遍历确定第一个元素a，然后查找使b+c = -a的b和c。

确定b和c时，只需要使用O(n)的算法，进行双向查找。如果使用哈希表需要对数组进行元素删除。为了不出现重复的序列，使用时将a对每个重复的元素只在第一次遇见时进行遍历。相同元素交换后仍然记为一组元素。

1. **class** Solution:
2. **def** threeSum(self, nums):
3. """
4. :type nums: List[int]
5. :rtype: List[List[int]]
6. """
7. length = len(nums)
8. nums.sort()
9. sums=[]
10. **for** i **in** range(length-2):
11. low,high =i+1,length-1
12. sumt = 0 - nums[i]
13. **if** (i==0) **or** (i > 0 **and** nums[i]!=nums[i-1]):
14. **while** low < high:
15. **if** nums[low]+nums[high]==sumt:
16. sums.append([nums[i],nums[low],nums[high]])
17. **while** (low < high) **and** (nums[low]==nums[low+1]):
18. low+=1
19. **while** (low < high) **and** (nums[high]==nums[high-1]):
20. high-=1
21. low+=1
22. high-=1
23. **elif** nums[low]+nums[high] < sumt:
24. low+=1
25. **else**:
26. high-=1
27. **return** sums

## [18. KSum (K=4)](https://leetcode.com/problems/4sum/description/)

Given an array *S* of *n* integers, are there elements *a*, *b*, *c*, and *d* in *S* such that *a* + *b* + *c* + *d* = target? Find all unique quadruplets in the array which gives the sum of target.

**Note:** The solution set must not contain duplicate quadruplets.

For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is:

[

[-1, 0, 0, 1],

[-2, -1, 1, 2],

[-2, 0, 0, 2]

]

思路：我们先使用双向遍历的方式解决2sum问题。然后递归解决3SUM…KSUM问题。

1. **def** findNsum(self, nums, target, N):
3. length = len(nums)
5. **if** length < N **or** N < 2:
6. **return** []
8. sums = []
9. **if** N == 2:
10. low, high = 0, length - 1
11. **while** low < high:
12. **if** nums[low] + nums[high] == target:
13. sums.append([nums[low], nums[high]])
14. **while** (low < high) **and** (nums[low] == nums[low + 1]):
15. low += 1
16. **while** (low < high) **and** (nums[high] == nums[high - 1]):
17. high -= 1
18. low += 1
19. high -= 1
20. **elif** nums[low] + nums[high] < target:
21. low += 1
22. **else**:
23. high -= 1
24. **return** sums
25. **else**:
26. **for** i **in** range(length - N + 1):
27. **if** target < nums[i] \* N **or** target > nums[-1] \* N:
28. **break**
29. **if** (i == 0) **or** (i > 0 **and** nums[i] != nums[i - 1]):
30. tmps = self.findNsum(nums[i + 1:], target - nums[i], N - 1)
31. **for** s **in** tmps:
32. sums.append([nums[i]] + s)
34. **return** sums

# DP

## 70. Climbing Stairs

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Note:** Given *n* will be a positive integer.

思路1

假设s[i]表示到达第i阶台阶时的方法数，由于可以从第i-1阶台阶或者第i-2阶台阶走到第i阶台阶。因此可以将原问题划分成规模更小的两个子问题。状态转移公式，如下所示：

*dp*[*i*]=*dp*[*i*−1]+*dp*[*i*−2]

1. **class** Solution:
2. **def** climbStairs(self, n):
3. """
4. :type n: int
5. :rtype: int
6. """
7. **if** n==1:
8. **return** 1
9. **if** n==2:
10. **return** 2
11. a=1
12. b=2
13. **for** i **in** list(range(3,n+1)):
14. b,a = a+b,b
16. **return** b

思路2：斐波那契数列

这题的状态转移方程是*dp*[*i*]=*dp*[*i*−1]+*dp*[*i*−2]等价于斐波那契数的计算方法*Fib*(*n*)=*Fib*(*n*−1)+*Fib*(*n*−2)。因此可以使用斐波纳挈计算公式进行求解。下式是特征方程的计算公式。*F*​*n*​​=1/√​5​​​((​2​​1+√​5​​​​​)​*n*+1​​−(​2​​1−√​5​​​​​)​*n*+1​​)

1. **public** **class** Solution {
2. **public** **int** climbStairs(**int** n) {
3. **double** sqrt5=Math.sqrt(5);
4. **double** fibn=Math.pow((1+sqrt5)/2,n+1)-Math.pow((1-sqrt5)/2,n+1);
5. **return** (**int**)(fibn/sqrt5);
6. }
7. }

## 53. Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [-2,1,-3,4,-1,2,1,-5,4],  
the contiguous subarray [4,-1,2,1] has the largest sum = 6.

思路和方法同 [Best Time to Buy and Sell Stock](#_121._Best_Time)

## 198. House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

思路:动态规划

S[i]表示从第i家商店离开时，所能获得的最大金额。对第i家商店可以选择抢或者不抢。如果不抢所能获得的金额就等于第i-1家商店的金额。如果抢所能获得的金额就是第i-2家商店的金额加上第i家抢的金额。

S[i] = max(S[i]-1, S[i-2]+p[i])

注意特数值的问题。

1. **class** Solution:
2. **def** rob(self, nums):
3. """
4. :type nums: List[int]
5. :rtype: int
6. """
7. length = len(nums)
8. **if** length == 0:
9. **return** 0
10. **if** length == 1:
11. **return** nums[0]
13. a = 0
14. b = 0
15. **for** i **in** range(length):
16. b,a = max(b,a+nums[i]),b
17. **return** b

## 303. Range Sum Query – Immutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

思路：这题更适合使用缓存的方法来处理，将数组中前i数(包括第i个数)的和存在s[i]，那么i到j之间的和就等于S[j]-S[i-1]。小技巧：在数组前面添加一个S[0]=0，这样就可以使用S S[j]-S[i],而不用注意下标了。

1. **class** NumArray(object):
2. **def** \_\_init\_\_(self, nums):
3. """
4. initialize your data structure here.
5. :type nums: List[int]
6. """
7. self.accu = [0]
8. **for** num **in** nums:
9. self.accu += self.accu[-1] + num,
11. **def** sumRange(self, i, j):
12. """
13. sum of elements nums[i..j], inclusive.
14. :type i: int
15. :type j: int
16. :rtype: int
17. """
18. **return** self.accu[j + 1] - self.accu[i]

## 338. Counting Bits

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

**Example:**  
For num = 5 you should return [0,1,1,2,1,2].

思路：以数字X为例子，假设X= 10011001。从二进制角度可以将它分解成两部分:最后的数字1和其他数字1001100，最后一个数字如果是1等价于这个数是奇数，前面的数字中的1的个数等于X>>1中的数字的个数。也就是说，d[i]表示数字i的二进制表示中数字1的个数。可以得到如下状态转移公式。

d[i] = d[i>>1]+(d&1)

1. **class** Solution:
2. **def** countBits(self, num):
3. """
4. :type num: int
5. :rtype: List[int]
6. """
7. bits = [0]\*(num+1)
8. **for** i **in** list(range(1,num+1)):
9. bits[i] = bits[i>>1]+(i&1)
10. **return** bits

## 416. Partition Equal Subset Sum

## 650. 2 Keys Keyboard

# Graph

## 542. 01 Matrix

Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell.The distance between two adjacent cells is 1.

思路一：使用BFS搜索:

首先将矩阵中值为0的点放入到队列中。然后依次出列，将附近的点的距离等于队列中的点加1。实现BFS的过程中使用Python中内置的Queue模块来实现。

1. **from** queue **import** Queue
2. **class** Solution:
3. **def** updateMatrix(self, matrix):
4. """
5. :type matrix: List[List[int]]
6. :rtype: List[List[int]]
7. """
8. R = len(matrix)
9. C = len(matrix[0])
10. ans = [[-1]\*C **for** i **in** range(R)]
11. q = Queue()
12. direct = [[-1,0],[1,0],[0,-1],[0,1]]#up down left right
13. **for** i **in** range(R):
14. **for** j **in** range(C):
15. **if** matrix[i][j]==0:
16. ans[i][j] = 0
17. q.put((i,j,0))
18. **while** **not** q.empty():
19. i,j,depth = q.get()
20. **for** k **in** range(4):
21. new\_i,new\_j = i+direct[k][0],j+direct[k][1]
22. **if** ((0<=new\_i **and** new\_i<R) **and** (0<=new\_j **and** new\_j<C)):
23. **if** (ans[new\_i][new\_j]<0):
24. ans[new\_i][new\_j] = depth+1
25. q.put((new\_i,new\_j,depth+1))
26. **elif**(ans[new\_i][new\_j]>depth+1):
27. ans[new\_i][new\_j] = depth+1
28. q.put((new\_i,new\_j,depth+1))
29. **return** ans

思路二:动态规划

如果Matrix[i][j]==0，那么这点的距离就是Distence[i][j]=0,否则

Distence[i][j] = min(Distence[i-1][j]+1, Distence[i+1][j]+1, Distence[i][j-1]+1, Distence[i][j+1]+1)

这样可以通过两次遍历的过程，完成上述过程。

第一次从左上角遍历到右下角。完成

dist[*i*][*j*]=min(dist[*i*][*j*],min(dist[*i*][*j*−1],dist[*i*−1][*j*])+1)

第二次从右下角遍历到左上角，完成

dist[*i*][*j*]=min(dist[*i*][*j*],min(dist[*i*][*j*+1],dist[*i*+1][*j*])+1)

1. **from** queue **import** Queue
2. **class** Solution:
3. **def** updateMatrix(self, matrix):
4. """
5. :type matrix: List[List[int]]
6. :rtype: List[List[int]]
7. """
9. R = len(matrix)
10. C = len(matrix[0])
11. **if** R==0 **or** C==0:
12. **return** matrix
14. ans = [[100000]\*C **for** i **in** range(R)]
15. **for** i **in** range(R):
16. **for** j **in** range(C):
17. **if** matrix[i][j] == 0:
18. ans[i][j] = 0
19. **else**:
20. ans[i][j] = min(ans[i][j],ans[i-1][j]+1,ans[i][j-1]+1)
22. **for** i **in** list(range(R-1,-1,-1)):
23. **for** j **in** list(range(C-1,-1,-1)):
24. **if** matrix[i][j]!=0:
25. **if** i<R-1:
26. ans[i][j] = min(ans[i][j],ans[i+1][j]+1)
27. **if** j<C-1:
28. ans[i][j] = min(ans[i][j],ans[i][j+1]+1)
30. **return** ans

# Stack

## 456. 132 Pattern

Given a sequence of n integers a1, a2, ..., an, a 132 pattern is a subsequence a**i**, a**j**, a**k** such that **i** < **j** < **k** and a**i** < a**k** < a**j**. Design an algorithm that takes a list of n numbers as input and checks whether there is a 132 pattern in the list.

**Note:** n will be less than 15,000.

思路一:通过遍历搜索来实现。

（第一轮循环）通过寻找最大元素nums[j],同时保留j前面最小元素。第二轮循环寻找第二大元素nums[k],如果发现满足条件的元素就查找成功。顺序遍历

1. **public** **class** Solution {
2. **public** boolean find132pattern(**int**[] nums) {
3. **int** min\_i = Integer.MAX\_VALUE;
4. **for** (**int** j = 0; j < nums.length - 1; j++) {
5. min\_i = Math.min(min\_i, nums[j]);
6. **for** (**int** k = j + 1; k < nums.length; k++) {
7. **if** (nums[k] < nums[j] && min\_i < nums[k])
8. **return** **true**;
9. }
10. }
11. **return** **false**;
12. }
13. }

思路二:利用栈存储所有比第三个元素大的元素。保证栈中是有序的且都比第三个元素大。

逆序遍历nums数组，如果发现元素比第三个原始小，就找到这样的模式了。否则从栈中弹出元素，更新第三个元素，并将第三个元素放入栈中。

1. **class** Solution:
2. **def** find132pattern(self, nums):
3. """
4. :type nums: List[int]
5. :rtype: bool
6. """
7. length = len(nums)
8. **if** length < 3:
9. **return** False
11. third = -float("inf")
12. liststack = list()
13. **for** i **in** reversed(range(len(nums))):
14. **if** third > nums[i]:
15. **return** True
17. **while** liststack **and** nums[i] > liststack[-1]:
18. third = liststack.pop()
19. liststack.append(nums[i])
21. **return** False

# Linked List

## Remove Nth Node From End of List

## Merge Two Sorted Lists

## 23. Merge k Sorted Lists

## 24. Swap Nodes in Pairs

## 25. Reverse Nodes in k-Group

## 61. Rotate List

# Hash Table

## 30. Substring with Concatenation of All Words

## 3. Longest Substring Without Repeating Characters