Solution 1

Method 1 (Simple) Use two loops. The outer loop runs from 0 to size – 1 and one by one picks all elements from left to right. The inner loop compares the picked element to all the elements to its right side. If the picked element is greater than all the elements to its right side, then the picked element is the leader.

In [16]:
```
# Python Function to print leaders in array

def printLeaders(arr,size):

    for i in range(0, size):
        for j in range(i+1, size):
            if arr[i]<=arr[j]:
                break
        if j == size-1: # If loop didn't break
            print (arr[i],end=' ')

# Driver function
arr=[7,10,4,10,6,5,2]
printLeaders(arr, len(arr))
```

```
10 6 5 2
```

Time Complexity: O(n*n)

Method 2 (Scan from right) Scan all the elements from right to left in an array and keep track of maximum till now. When maximum changes its value, print it.

In [13]:
```
# Python function to print leaders in array
def printLeaders(arr, size):

    max_from_right = arr[size-1]
    print (max_from_right,end=' ')
    for i in range( size-2, -1, -1):
        if max_from_right < arr[i]:
            print (arr[i],end=' ')
            max_from_right = arr[i]

# Driver function
arr=[7,10,4,10,6,5,2]
printLeaders(arr, len(arr))
```

```
2 5 6 10
```

Time Complexity: O(n)

Solution 2

We need to find out the maximum difference (which will be the maximum profit) between two numbers in the given array. Also, the second number (selling price) must be larger than the first one (buying price).

In formal terms, we need to find \max(\text{prices[j]} - \text{prices[i]})max(prices[j]−prices[i]), for every ii and jj such that j > ij>i.

Approach 1: Brute Force

class Solution: def maxProfit(self, prices: List[int]) -> int: max_profit = 0 for i in range(len(prices) - 1): for j in range(i + 1, len(prices)): profit = prices[j] - prices[i] if profit > max_profit: max_profit = profit return max_profit Complexity Analysis Time complexity: O(n^2) Loop runs n(n−1)/2 times. Space complexity: O(1)O(1). Only two variables - maxprofit and profit are used. Approach 2: One Pass Algorithm Say the given array is: [7, 1, 5, 3, 6, 4] class Solution: def maxProfit(self, prices: List[int]) -> int: min_price = float('inf') max_profit = 0 for i in range(len(prices)): if prices[i] < min_price: min_price = prices[i] elif prices[i] - min_price > max_profit: max_profit = prices[i] - min_price return max_profit Complexity Analysis Time complexity: O(n)O(n). Only a single pass is needed. Space complexity: O(1)O(1). Only two variables are used.

Solution 3

class Solution: def subsetXORSum(self, nums: List[int]) -> int:

```
    """
    an efficient approach: find bitwise OR of array. ith bit
contribution to to xor sum is 2**(n-1).
    The result is 2**(n-1) * bitORSum

    Explanation:

    consider ith bit for the final result. If there is no element
whose ith bit is 1, then all xor subset
    sums has 0 in ith bit; if there are k (k>=1) elements whose ith
bits are 1, then there are in total
    comb(k, 1) + comb(k,3) + .. = 2**(k-1) ways to select odd number
our of these k elements to make the subset
    xor sum's ith bit to be 1, and there are 2**(n-k) ways to choose
from the remaining elements whose ith bits
    are 0s. Therefore, we have in total 2**(k-1) * 2**(n-k) = 2**(n-1)
subsets whose xor sums have 1 in their
    ith bit, which means the contribution to the final sum is 2**(n-1)
* 2**i. Notice this result is irrelevant
    to k.

    So we only need to determine whether there is any element whose
ith bit is 1. We use bitwise OR sum to do this.

    Time complexity: O(n), Space: O(1) """
```

bits = 0 for a in nums: bits |= a return bits * int(pow(2, len(nums)-1))

In [ ]: