105. Construct Binary Tree from Preorder and Inorder Traversal
We will use recursion to build TreeNode, each time we will separate inorder and preorder array. TC is O(n * n)

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if not preorder:
            return None
        root = TreeNode(preorder[0])
        index = inorder.index(preorder[0])
        inorder_left = inorder[:index]
        inorder_right = inorder[index + 1:]
        preorder_left = preorder[1:len(inorder_left) + 1]
        preorder_right = preorder[len(inorder_left) + 1:]
        root.left = self.buildTree(preorder_left, inorder_left)
        root.right = self.buildTree(preorder_right, inorder_right)
        return root
```

2. Random Pick with Weight
We will accumulate weights from head to end. And then we will use binary search to get the random number from 1 to largest weight. Get largest number that less than or equal to the random number. In the end, we will return its index. TC is O(logn)

```
from bisect import *
from random import *
class Solution:

    def __init__(self, w: List[int]):
        self.weights = []
        cur_sum = 0
        for i in w:
          cur_sum += i
          self.weights.append(cur_sum)
        self.total = cur_sum

    def pickIndex(self) -> int:
        return bisect_left(self.weights, randint(1, self.total))
```

239. Sliding Window Maximum
We will use a deque to manage our number, if len(deque) == k, we will popleft(), if the pop number is max, we will find a new cur_max from deque, then we will compare the new number with cur_max, if it's larger or equal to cur_max, we will clear deque, and let cur_max = num, and append num to memo. TC is O(kn)

```
from collections import deque
class Solution:
```

```
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums:
          return []
        if k == 1:
          return nums
        memo = deque(nums[:k])
        cur_max = max(memo)
        result = [cur_max]
        for num in nums[:k]:
          if num >= cur_max:
            cur_max = num
            memo.clear()
          memo.append(num)
        for num in nums[k:]:
          if len(memo) == k:
            temp = memo.popleft()
            if temp == cur_max:
              cur_max = max(memo)
          if num >= cur_max:
            cur_max = num
            memo.clear()
          memo.append(num)
          result.append(cur_max)
        return result
```

4. Maximum Subarray

We will accumulate all numbers from left to right, so every sub sum from j - 1 to i is nums[i] - nums[j], so we only need to maintain our cur_min and cur_max in every iteration. In the end, we will return cur_max

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        cur_min = min(nums[0], 0)
        cur_max = nums[0]
        for i in range(1, len(nums)):
          nums[i] += nums[i - 1]
          cur_max = max(cur_max, nums[i] - cur_min)
          cur_min = min(cur_min, nums[i])
        return cur_max
```

5. Maximum Product Subarray

We will always remember cur_min and cur_max. And in each iteration, we will do result = (result, cur_max * num, cur_min * num). TC is O(n)

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

```
cur_min = min(1, nums[0])
cur_max = max(1, nums[0])
result = nums[0]
for num in nums[1:]:
  num1, num2 = num * cur_min, num * cur_max
  result = max(result, num1, num2)
  cur_min = min(1, num1, num2)
  cur_max = max(1, num1, num2)
return result
```