Subarray Problems

209. Minimum Size Subarray Sum (https://leetcode.com/problems/minimum-size-subarray-sum/)

Medium

Given an array of n positive integers and a positive integer s, find the minimal length of a contiguous subarray of which the sum ≥ s. If there isn't one, return 0 instead.

Example:

```
Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
```

Explanation: the subarray [4,3] has the minimal length under the problem constraint.

Follow up:

If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n).

Solution

We use a sliding window approach with pointers to either side of the window, moving it along until we have covered the whole array.

- Start with left and right pointers to the indices of the required subarray both zero.
- Set the initial min subarray length to 1 more than the length of the array nums.
- While the right pointer is within the array nums:
  - If the sum of the current subarray is less than s, move the right pointer to the right
  - If the sum of the current subarray is more than s,
    - record the subarray length as minimal if it is smaller than the current
    - move the left pointer to the right

Performance

- Runtime: 48 ms, faster than 77.62% of Python3 online submissions for Minimum Size Subarray Sum.
- Memory Usage: 13.9 MB, less than 100.00% of Python3 online submissions for Minimum Size Subarray Sum.

Complexity Analysis

```
O(n) in time
O(1) in space
```

In [1]:
```
class Solution:
    def minSubArrayLen(self, s: 'int', nums: 'List[int]') -> 'int':
        n = len(nums)
        min_len = n + 1
        left, right = 0, 0
        sub_sum = 0
        while right < n:
            sub_sum += nums[right]
            #print('l=', left, 'r=', right, 'min_len=', min_len, 'sum=', sub_sum, nums[
            if sub_sum >= s:
                min_len = min(min_len, right - left + 1)
                sub_sum -= nums[left]
                left += 1
                sub_sum -= nums[right]
            else:
                right += 1
        return min_len if min_len < n + 1 else 0

my_sol = Solution()
print('Should print 2:', my_sol.minSubArrayLen(7, [2,3,1,2,4,3]))
print('Should print 0:', my_sol.minSubArrayLen(100, []))
print('Should print 5:', my_sol.minSubArrayLen(15, [1,2,3,4,5]))
```

```
Should print 2: 2
Should print 0: 0
Should print 5: 5
```

53. Maximum Subarray (https://leetcode.com/problems/maximum-subarray/)

Easy

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

```
Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4],
Output: 6
```

Explanation:

```
[4, -1, 2, 1] has the largest sum = 6.
```

Follow up:

If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Note: The O(n) algorithm is a well known dynamic programming solution known as Kadane's algorithm

Solution

- Kadane's Algorithm (https://en.wikipedia.org/wiki/Maximum_subarray_problem)

Performance

- Runtime: 48 ms, faster than 65.24% of Python3 online submissions for Maximum Subarray.
- Memory Usage: 13.7 MB, less than 5.50% of Python3 online submissions for Maximum Subarray.

Complexity Analysis

- O(n) in time.
- O(1) in space.

In [2]:
```
class Solution:
    def maxSubArray(self, nums: 'List[int]') -> 'int':
        max_ending_here = max_so_far = nums[0]
        for i in range(1, len(nums)):
            # either start at the next index or add the value at the next index to the
            max_ending_here = max(nums[i], max_ending_here + nums[i])
            max_so_far = max(max_so_far, max_ending_here)
        return max_so_far

my_sol = Solution()
print('Should print 6:', my_sol.maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4]))
```

```
Should print 6: 6
```

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

Easy

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Example 1:

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

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 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
```

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

Solution

- Start with max profit assumed to be zero
- We traverse the array from left to right, at each index calculating
  - the minimum of the subarray which ends at the current index
  - the profit we would make if we sold at the current value (having bought at the previous minimum price)
  - the maximum profit we could have made over all the previous indices

Performance

- Runtime: 48 ms, faster than 52.11% of Python3 online submissions for Best Time to Buy and Sell Stock.
- Memory Usage: 14 MB, less than 5.08% of Python3 online submissions for Best Time to Buy and Sell Stock.

Complexity Analysis

- O(n) in time.
- O(1) in space.

In [3]:
```
class Solution:
    def maxProfit(self, prices: 'List[int]') -> 'int':
        if len(prices) > 1:
            min_price = prices[0]
            max_profit = 0
            for i in range(1, len(prices)):
                min_price = min(min_price, prices[i])
                max_profit = max(max_profit, prices[i]-min_price)
            return max_profit

my_sol = Solution()
print('Should print 16:', my_sol.maxProfit([45, 24, 35, 31, 40, 38, 11]))
print('Should print 5:', my_sol.maxProfit([7,1,5,3,6,4]))
print('Should print 0:', my_sol.maxProfit([7,6,4,3,1]))
```

```
Should print 16: 16
Should print 5: 5
Should print 0: 0
```

152. Maximum Product Subarray (https://leetcode.com/problems/maximum-product-subarray/)

Medium

Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.

Example 1:

```
Input: [2,3,-2,4]
Output: 6
```

Explanation: [2,3] has the largest product 6.

Example 2:

```
Input: [-2,0,-1]
Output: 0
```

Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

Performance

- untime: 48 ms, faster than 72.42% of Python3 online submissions for Maximum Product Subarray.
- Memory Usage: 13.4 MB, less than 21.55% of Python3 online submissions for Maximum Product Subarray.

Complexity Analysis

- O(n) in time.
- O(1) in space.

In [4]:
```
class Solution:
    def maxProduct(self, nums: 'List[int]') -> 'int':
        if len(nums) > 0:
            # need to track min and max since nums[i] can be negative
            min_ending_here = max_ending_here = max_so_far = nums[0]
            for i in range(1, len(nums)):
                min_ending_prev = min_ending_here
                # The min/max ending at i can either be the result of multiplying nums[
                # or simply nums[i]
                min_ending_here = min(max_ending_here*nums[i], min_ending_prev*nums[i],
                max_ending_here = max(max_ending_here*nums[i], min_ending_prev*nums[i],
                max_so_far = max(max_so_far, max_ending_here)
            return max_so_far

my_sol = Solution()
print('Should print 6:', my_sol.maxProduct([2,3,-2,4]))
print('Should print 0:', my_sol.maxProduct([-2,0,-1]))
```

```
Should print 6: 6
Should print 0: 0
```

76. Minimum Window Substring (https://leetcode.com/problems/minimum-window-substring/)

Hard

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

Example:

```
Input: S = "ADOBECODEBANC", T = "ABC"
Output: "BANC"
```

Note:

- If there is no such window in S that covers all characters in T, return the empty string "".
- If there is such window, you are guaranteed that there will always be only one unique minimum window in S.

Solution

We use a sliding window approach with pointers to either side of the window, moving it along until we have covered the whole array.

- Start with left and right pointers to the indices of the required substring both zero.
- Set the initial min substring length to 1 more than the length of the string S.
- While the right pointer is within the string S:
  - If the current substring doesn't contain all the required letters, move the right pointer to the right
  - If the current substring does contain all the required letters,
    - record the substring length as minimal if it is smaller than the current
    - move the left pointer to the right

Performance

- Runtime: 184 ms, faster than 30.06% of Python3 online submissions for Minimum Window Substring.
- Memory Usage: 13.4 MB, less than 17.22% of Python3 online submissions for Minimum Window Substring.

Complexity Analysis

- O(n + m) in time.
- O(m) in space.

In [5]:
```
from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        ns, nt = len(s), len(t)
        #print('ns =', ns, 'nt =', nt)
        if nt <= ns:
            # minW = (win_len, left_idx, right_idx)
            minW = (ns + 1, 0, ns)
            t_dict = Counter(t) # number of letters remaining to find
            t_remaining = nt    # number of letters remaining to find
            left, right = 0, 0
            check_right = True
            while right < ns:
                #print(s[left:right+1])
                if check_right:
                    if s[right] in t_dict:
                        t_dict[s[right]] -= 1
                        if t_dict[s[right]] >= 0:
                            t_remaining -= 1
                    #print('t_dict', t_dict, t_remaining)
                    if t_remaining == 0:
                        minW = min(minW, (right - left + 1, left, right + 1))
                        #print('minW = ', minW[0])
                        if minW[0] == nt:
                            return s[minW[1]:minW[2]]
                        if s[left] in t_dict:
                            t_dict[s[left]] += 1
                            if t_dict[s[left]] > 0:
                                t_remaining += 1
                        left += 1
                        check_right = False
                    else:
                        right += 1
                        check_right = True
                if minW[0] < ns + 1:
                    return s[minW[1]:minW[2]]
        return ''

my_sol = Solution()
print('Should print BANC:', my_sol.minWindow("ADOBECODEBANC", "ABC"))
print('Should print a:', my_sol.minWindow("a", "a"))
```

```
Should print BANC: BANC
Should print a: a
```

In [ ]: