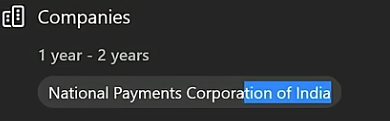
# NUMBERS

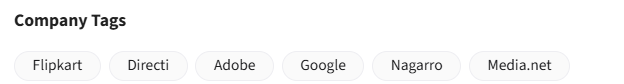
# Arrays

Q1. [**Special Array I**](https://leetcode.com/problems/special-array-i/)



# Hashing

Q1. [**Valid Anagram**](https://leetcode.com/problems/valid-anagram/)



#O(nlog(n)) approach

class Solution:

    def isAnagram(self, s: str, t: str) -> bool:

        s = sorted(list(s))

        t = sorted(list(t))

        return s==t

#using hashing

class Solution:

    def isAnagram(self, s: str, t: str) -> bool:

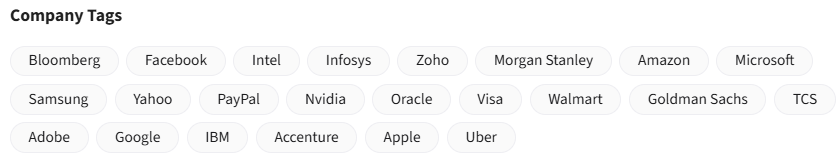
        cs = Counter(s)

        ct = Counter(t)

        return cs==ct

# DYNAMIC PROGRAMMING

Q1. [**Best Time to Buy and Sell Stock**](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/)



class Solution:

    def maxProfit(self, prices: List[int]) -> int:

        min\_price = prices[0]

        profit=0

        for price in prices[1:]:

            profit = max(profit, price-min\_price)

            min\_price = min(min\_price, price)

        return profit

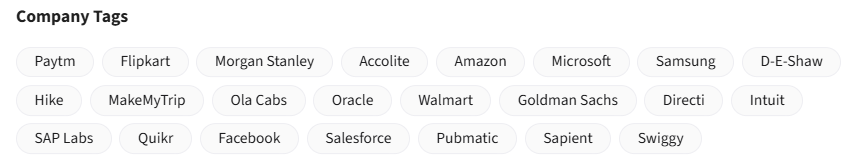
We want to **maximize profit by buying and selling the stock once** (buy before sell). To solve this in a single pass:

**1** . Initialize min with the highest possible value (Integer.MAX\_VALUE) to keep track of the **lowest price (best buying point)** so far.  
**2** . Initialize maxProfit to 0 to track the **maximum profit** encountered.  
**3** . Loop through the prices array:

* Update min to be the smaller of the current price and the previous min.
* Calculate the profit if we sell at the current price: price - min.
* Update maxProfit if this profit is greater than the previous maxProfit.

This **greedy approach** ensures we always make the most profitable decision at each point without needing to check **every pair**.

Q2. [**Best Time to Buy and Sell Stock II**](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/)



It adds up profits **every time there's a price increase**, assuming you buy at the previous day's price and sell at today's higher price.  
By updating buy\_p to the current price on each iteration, it tracks local buying points and adds profits for every upward movement.

class Solution:

    def maxProfit(self, prices: List[int]) -> int:

        profit = 0

        buy\_p = prices[0]  #buying price

        for price in prices[1:]:

            if price>buy\_p:

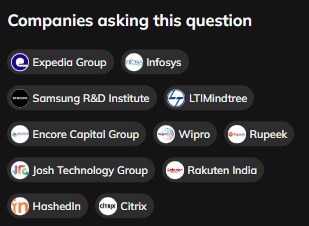
                profit += price-buy\_p

            buy\_p = price

        return profit

# Heap

Q1. [**Last Stone Weight**](https://leetcode.com/problems/last-stone-weight/)



If we always smash the two **heaviest stones**, we minimize leftover weights.  
A **max-heap** is the perfect data structure to repeatedly access and remove the largest stones efficiently.

* Push all stone weights into a **max-heap**.
* While more than one stone remains:
  + Pop the two heaviest stones.
  + If their weights differ, push the difference back into the heap.
* When done, return the last stone's weight (or 0 if none left).

class Solution:

    def lastStoneWeight(self, stones: List[int]) -> int:

        #since python supports only min heap, we can negate all values to act as max heap

        for i in range(len(stones)):

            stones[i] = - stones[i]

        heapq.heapify(stones)

        #simulation

        while(len(stones)>1):

            max1 = heapq.heappop(stones)

            max2 = heapq.heappop(stones)

            if max1!=max2:

                heapq.heappush(stones, max1-max2)

        return -stones[0] if stones else 0