- Arrays :-

- **Majority Element (Moore's voting algorithm) :-**

Time complexity → O(N) → space (O)(1)

1	4	3	2	4	4	4	2

↑

pseudo code ⟶

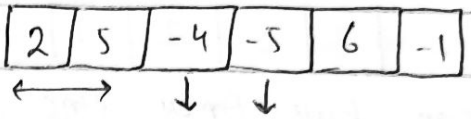
```
element = array[0]
count = 1;
if (element != array[i])
    count --;
    if (count == 0)
        count = 1;
        element = array[i]
else if (element = array[i]
    count++
```

Here we check the next element if it is not then we use decrement.

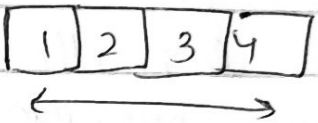
→ This Problem can be solved using map.
But it time complexity is O(N) ⟶ space O(N).

• Largest Sum Contiguos SubArray (Kadane's algorithm) :-

- if a negative number occurs we move to the next element and store sum.

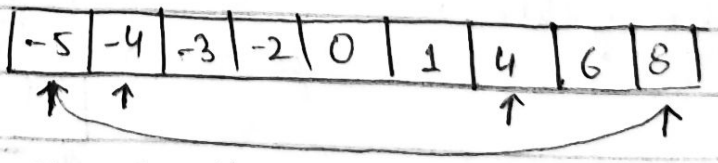



```
function largestSum (array)
{
    let maxSum = 0, curSum = 0;
    for (let i=0; i< array.length; i++)
    {
        curSum = curSum + array[i]
        if (curSum > maxSum)
            maxSum = curSum;
        if (curSum < 0)
            cu = 0;
    }
    return maxSum;
}.
```



Steps → 3 ; → maxSum
→ 1
→ 6 → maxSum
→ 12 → maxSum
→ 9

• Check sum zero (array must be sorted) :-

a = 

Approach :-

Left = 0 ; Right = array.length - 1 , sum ;
sum = a[left] + a[right]
if sum > 0 right --
else left ++ ;
sum == 0 ; console.log.

- **Best time to Buy & sell stocks → LeetCode - 121.**

3	5	1	7	4	9	3

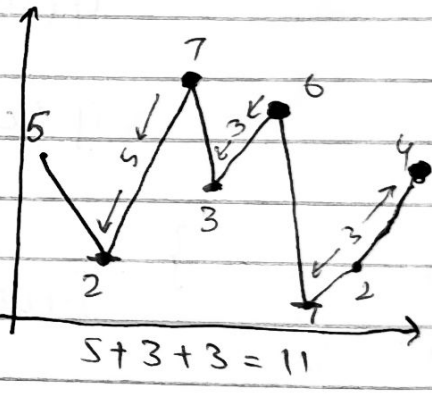
You can buy only one time ∴

```
min = Math.min(min, a[i]);
max = Math.max(max, a[i]);
after loop the return
return max - min;
```

- **Best time to Buy & Sell Stocks → LeetCode 122 :-**

5	2	7	3	6	1	2	4

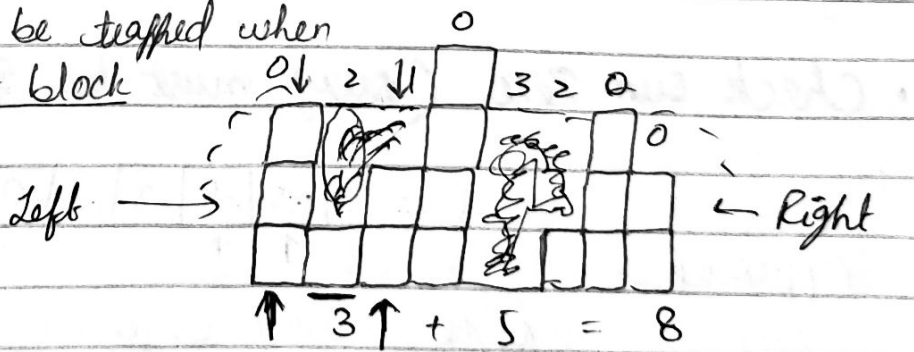
```
Let profit = 0;
if (array[i] > array[i-1])
    profit += array[i] - array[i-1];
return profit;
```



- **Trapping Rain water - LeetCode 42 :-**

3	1	2	4	0	1	3	2

Logic :- water will only be trapped when Left block and right block will be greater.



Left-Max = 3, 3, 3, 4, 4, 4, 4, 4
Right-Max = 4, 4, 4, 4, 3, 3, 3, 2
value = Min(Left[i], right[i]) - a[i]

- **Sorting Algorithms :-**

→ **Bubble Sort :-**

Swapping is used and the largest element will move to the end.

8	4	1	5	9	2

in first iteration → i = n → j = 0 ; j <= i ; j++

if (array[j] > array[j+1])

swap (array[j], array[j+1])

4	8	1	5	9	2
4	1	5	8	9	2

4	1	8	5	9	2
4	1	5	8	2	9

→ Largst at end.

- **insertion Sort :-**

Take array and split it in two parts

8	4	1	5	9	2

↑ sorted ↑ non-sorted.

Store element in temp =

```
for (let i = 1; i < n; i++)
    let temp = a[i];
    let j = i-1;
    while (j >= 0 && a[j] > temp){
        array[j+1] = array[j]
        j--;
    }
    array[j+1] = temp;
```

	8	1	5	9	2

bcs of while loop we incremented one time.

temp

8	4	1	5	9	2

4	8	1	5	9	2

1	4	8	5	9	2

1	4	5	8	9	2

1	4	5	8	9	2

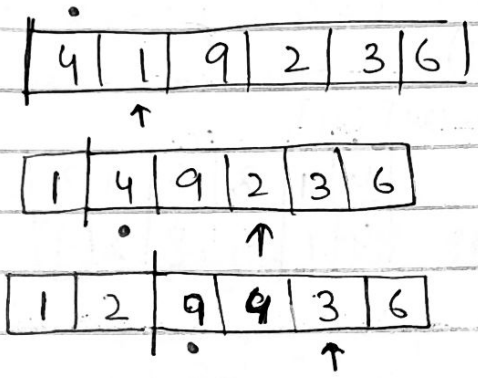
1	2	4	5	8	9

- **Insert Selection Sort :-**

  Take min value and arrange it / swap it with first value

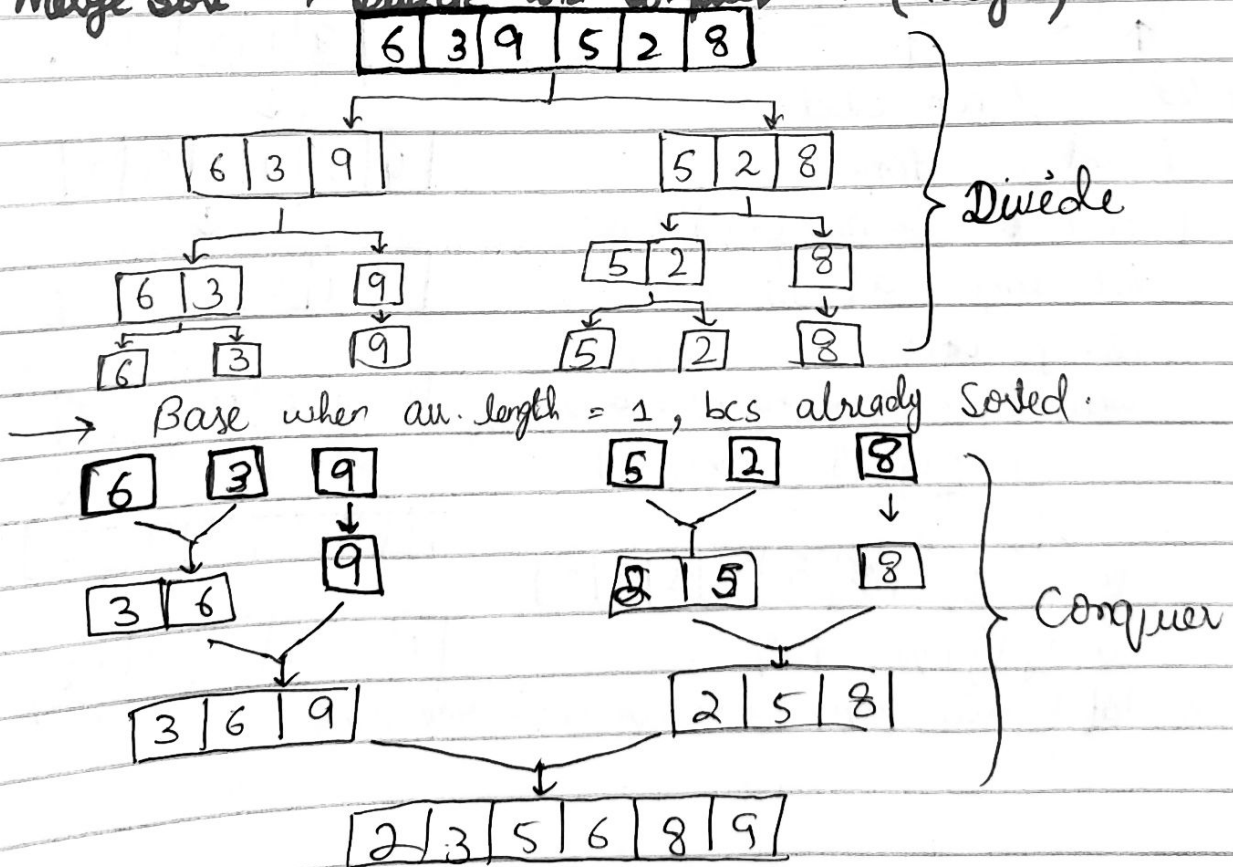
  Outer loop runs $< n-1$ times.

  Inner loop runs $< n$ times.



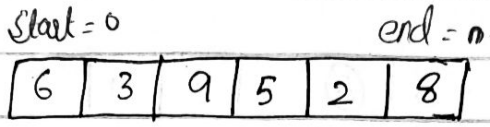
Dot represents min value and arrow j^{th} value.

- **Merge Sort → Divide and Conquer → (nlogn)**



• Quick Sort :-

Pivot and Partion



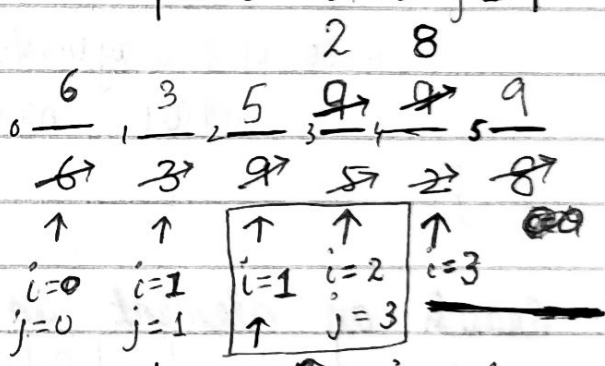
i = start - 1 ;

Pivot
- random
- mid
- first
- last

Traversing

```
for (let j = start ; j < end ; j++){
    if ( array[j] < pivot )
        i++
        Swap ;
    }
i++
swap (array[end], array[i]);
return i;
}.
```

Loop will end at j = 4



Swaped when i reached at the index - 1 times from higher position

```
function quickSort (array)
{
  pivotIndex = partition(a[], start, end);

  quickSort (array, start, pivot - 1);
  quickSort (a[], pivot+1, end);
}.
```