Arrays / strings.

1. Merge strings Alternatively:

input:

word1 : 'abc'

word2 : 'pqr'

output:

"apbqcr"

Ans:

```
def merge(word1, word2):

    res = ""
    l1, l2 = len(word1), len(word2).
    for i in range(min(l1, l2)):
        res += word1[i]
        res += word2[i]
    if l1 > l2:
        res += word1[l1:]
    else:
        res += word2[l2:]
    return res.
```

T(o) = O(Min(len(word1), len(word2))

space = O(1)

* Greatest Common Divisor of strings:

Input: str 1 = "ABC ABC" i/p: "ABAB AB" = str1

str 2 = "ABC" str2 = "ABAB"

output: "ABC o/p: "AB"

return longest string that divides both str1 & str2.

```
def gcdofstrings (str1, str2):

    if str1 + str2 != str2 + str1:
        return ""

    def gcd(l1, l2):
        while l2:
            l1, l2 = l2, l1 % l2
        return l1
    return str1[: gcd[len(str1), len[str2]]
```

→ Euclids Algorithm

Eg: l1 = 48, l2 = 18.

48, 18
18, 12
12, 6
6, 0.

* GCD of two numbers:

```
def gcd (l1, l2):
    while l2:
        l1, l2 = l2, l1 % l2
    return l1.
```

3. Can place the flowers:

input : flowerbed = [1, 0, 0, 0, 1]
n = 1

o/p : True

→ No flowers can be placed adjacent.

(2)
[1, 0, 0, 0, 1]
n = 2
false.

0 → empty
1 → full.

```
def canplaceflowers(flowerbed, n):
    c = 0
    for i in range(len(flowerbed)):
        if flowerbed[i] == 0:
            left = (i==0) or (flowerbed[i-1] == 0)
            right = (i == len(flowerbed)-1) or (flowerbed[i+1]==0)

            if left and right:
                flowerbed[i] = 1
                c += 1
                if c >= n:
                    return True

    return c >= n
```

4. Reverse vowels of a string!

* Two pointers

i/p: "IceCream"

o/p: "AceCreIm"

```
def reversevowels(s):
    l = list(s)
    i = 0                    ~> start
    j = len(s)-1             ~> end.
    vowels = "aeiouAEIOU"
    while i<j:
        while i<j and s[i] not in vowels:
            i+=1
        while i<j and s[j] not in vowels:
            j-=1
        l[i], l[j] = l[j], l[i]   ~> Swapping.
        i+=1
        j-=1
    res = "".join(l)
    return res.
```

* Reverse words in a string!

i/p : s = " the sky is blue "

o/p : " blue is the sky ".

```
def reverse words (s):
    l = s. split(" ")
    return " ". join (l[::-1])
```

Two Pointer

→ Time : O(n)
Space : O(n)

```
def reversewords (s):
    l = s. split()
    left = 0
    right = len (s) - 1

    while left < right :
        l[left], l[right] = l[right], l[left]
            left += 1
            right -= 1
    return " ". join(l).
```

* Product of Array Except Self:

? Sum Prefix concept.

i/p: [1, 2, 3, 4]

o/p: [24, 12, 8, 6]

```
def productExceptSelf (nums):

    res = [1] * len(nums)

    left = 1
    for i in range(len(nums)):
        res[i] = res[i] * left
        left = left * nums[i]

    right = 1
    for i in range(len(nums)-1, -1, -1):
        res[i] *= right
        right *= nums[i]

    return res.
```

→ iteration from the left.

→ to start the iteration from the end.

Greedy Approach

* Increasing triplet subsequence.

i/p : [1, 2, 3, 4, 5]
o/p : True

~> there should be a triplet where, $l[i] < l[j] < l[k]$
$i < j < k$.

```
def increasingtriplet(nums):

    first = float('inf')
    second = float('inf')        } ~> infinite +ve number.

    for n in nums:
        if n <= first:
            first = n
        elif n <= second:
            second = n
        else:
            return True
    return false.
```

* Two Pointers *

→ Move zeroes to the end:

i/p : [0, 1, 0, 3, 12]

o/p : [1, 3, 12, 0, 0]

```
def movezeroes(nums):
    left = 0

    for right in range(len(nums)):
        if nums[right] != 0:
            nums[right], nums[left] = nums[left], nums[right]
            left += 1

    return nums.
```

Work-flow:

[0, 1, 0, 3, 12]
 L
 R

[0, 1, 0, 3, 12]
 L R

[1, 0, 0, 3, 12]
    L R

[1, 0, 0, 3, 12]
    L   R

[1, 3, 0, 0, 12]
       L   R

[1, 3, 12, 0, 0]
          L R

* Is Subsequence:

i/p : s = "abc", t = "ahbgdc" } → they should be in the same order.

o/p : True

```
def isSubsequence(s, t):
    i = j = 0
    while i < len(s) and j < len(t):
        if s[i] == t[j]:
            i += 1
        j += 1
    return i == len(s)
```

workflow:

→ start two pointers each at both the strings.

→ Increment one after one, after each element matches.

* Container with most water:

i/p : height = [1, 8, 6, 2, 5, 4, 8, 3, 7]

o/p : 49.

```
def maxArea ( height ) :
    max_area = 0
    left = 0
    right = len (height) - 1
    while left < right:
        max_area = max( max_area, (right - left) *
                                    min (height[left], height[right])
        if height [left] < height [right] :
            left += 1
        else :
            right -= 1
    return max_area.
```

* Max Number of k-Sum pairs:

i/p: nums = [1,2,3,4]

k = 5

o/p = 2

```
def maxpairs(nums, k):
    nums.sort()

    i = 0
    j = len(nums) -1
    res = 0

    while i < j:
        if (nums[i] + nums[j]) == k:
            res += 1
            i += 1
            j -= 1
        elif (nums[i] + nums[j]) < k:
            i += 1
        else:
            j -= 1
    return res.
```

* Sliding Window *

* Maximum Average Subarray I :

i/p : [1, 12, -5, -6, 50, 3] , k = 4

o/p : 12.75

~> find the max_avg of subarray with length 'k'.

```
def findmaxAvg(nums, k):
    currsum = maxsum = sum(nums[:k])

    for i in range(k, len(nums)):
        currsum = currsum + nums[i] - nums[i-k]
        maxsum = max(currsum, maxsum)

    return maxsum/k.
```

8 patterns in the leetcode:

8. Sliding window
7. Subset pattern
6. Modified Binary search.
5. Top k elements → (Heap)
4. Binary Tree DFS.
3. Topological sort
2. Binary Tree BFS.
1. Two pointers.