DSA - logics

① Arrays

① min & max elements :

① Iterating every value in the array check them it is min or max with initialized & updated values.

② Reverse An array :

① iterating the array, until this condition (i >= len(arr) - i - 1) not satisfies (we go until half of array) it helps to swap front & back values.

③ Maximum_Subarray.py :

① kadane's Algo.

Se while iterating adding all the elements to a sum variable & checking with previous sum & current_sum, largest will be stored & returned at last.

④ Contains_Duplicates :

~~Storing the~~ Converting the given array to 'set' & to list, Comparing both lengths if equal no duplicates, otherwise Contain.

⑤ Chocolate distributions

Sorting the arr (no. of chocolate packets) taking m sized subarrays & finding the diff blw max & min values of subarrays, storing them & returning min difference.

⑥ Search in Rotated Sorted

→ We know left is sorted, so the Binary Search technique can be used. If target = arr[mid] return
→ we have to check in left half when (arr(low) <= arr(mid)), if element (target) present in low <= target <= mid, high = mid - 1, or else low = mid + 1.
→ If a[low] => arr[mid], got to right half check its presence. mid <= target <= arr[high] low = mid + 1 or else high = mid - 1.

⑦ next permutation

① Pivot should be find by using this Condition $A[i] < A[i+1]$

② Find the right most element that should just greater than pivot.

③ Reverse the elements just above the pivot elements. i.e (pivot +1) to n-1

⑧ Best time to Sell Stock

We have to iterate through the array and should store the min value till index(i), then we need to substract the cost_value with min_value.

⑨ Missing & Repeating

→ ~~Missing~~ Repeating

To find ~~m~~missing we iterate through list, take abs(val) and check with nums[val-1] is >0. if it make it '-ve' otherwise it is over repeating value.

val = nums[i]

→ Missing

Now iterate the array which is the result of repeating processes if you find any positive value, then missing will be index +1.

(10) k-th Largest Element in an Array

→ Create a list → heap, iterate list & push to heap, check the len. after pushing, if the len(heap) > k, pop the value.

return heap[0].

(11) Trapping Rain Water

→ Take 2 pointers and 2 variable lmax, rmax. these are use to store max values of left & right.

→ we need to do this `min(lmax, rmax) - height[i]`

(12) Product of Array Except Self.

→ Create a result_list with size len(nums).

→ prefix = 1
iterate upto length of the give list.
result[i] = prefix
now, prefix *= nums[i]

→ Postfix = 1
iterate from reverse
result[i] *= postfix
postfix *= nums[i] # updating Postfix

(13) Maximum Product Subarray.

→ iterate through the array, store the product in one variable (s=1) by comparing the product(s) with previous product.

```
if s i==0
    s=1
```

→ iterate the array from back
  ↳ store the product in (s=1)
    ↳ compare the cur-product with previous

```
if s ==0:
    s=1
```

(14) Find minimum in Rotated Sorted Array :

→ Take 2 pointers and a variable name → min_element

→ use Binary Search.

```
↳ nums[l] < nums[h]  # checks ascending order
    min_element = min(min_element, nums[low])
    break
→ nums[low] :z= nums[mid]  # checks first half
    min_element = min(min_element, nums[mid])
    low = mid+1
→ high = mid-1
    min_element = min(min_element, nums[mid])
return min_element.
```

(15) Pair Sum in a Sorted & Rotated Array:

Two pointers.

→ initialize i = 0

→ iterate the array

↳ arr[i] > arr[i+1]

i = i+1

break.

→ let r = i & l = (i+1) % len(arr)

→ iterate till (l != r) satisfies

↳ arr[l] + arr[r] == target:

return True

↳ arr[l] + arr[r] < target

l = (l+1) % len(arr)

↳ r = (r-1) % len(arr)

→ return False.

(16) Container with most water (finding rectangle area)

→ using Two pointers. l = 0, r = len(nums) - 1

→ l < r:

↳ area = (r-l) * min(height[l], height[r])

↳ storing max value in res

if nums[l] < nums[r]

l += 1

else:

r -= 1

7) 3 Sum

→ initialize a list res = [ ]
→ Sort(nums)
→ iterate through the list
  ↳ i>0 & nums[i] == nums[i-1]:
    continues.
  ↳ j = i+1
    k = len(nums) - 1
  ↳ while j < k :
    ↳ total = nums[i] + nums[j] + nums[k]
    ↳ if total > 0 :
      k -= 1
    ↳ elif total < 0:
      j += 1
    ↳ else:
      res.append([nums[i], nums[j], nums[k])
      j += 1
      while nums[j] == nums[j-1] & j < k :
        j += 1

return res.

(18) kth smallest

→ initializing a list. (max_heap)

→ iterating given list.

↳ pushing into heapq

heapq.heappush(max_heap, -num)

↳ if len(max_heap) > k

heapq.heappop(max_heap)

return -max_heap[0]

(19) Merge overlap intervals (gives a 2D array)

→ Sort the array/list & taking res list to store

→ iterating the array list

↳ start = arr[i][0]

end = arr[i][1]

↳ checking & skipping already ~~sorted~~ merged interval

if res and res[-1][1] >= end:

continue

↳ iterating from i+1 → n

if arr[j][0] <= end:

end = max(end, arr[j][1])

add to res (start, end)

(20) Find the minimum no. of merge operations to make an array palindrome

nums is a input array

→ keep a tracker, (i, j) 0, len(nums)-1

→ two pointers

↳ if first & last elements equal

i += 1

j -= 1

nums[i] > nums[j]

j -= 1

arr[j] += arr[j+1] (adds two elements)

tracker += 1

nums[i] < nums[j]

i += 1

arr[i] += arr[i-1]

ans += 1