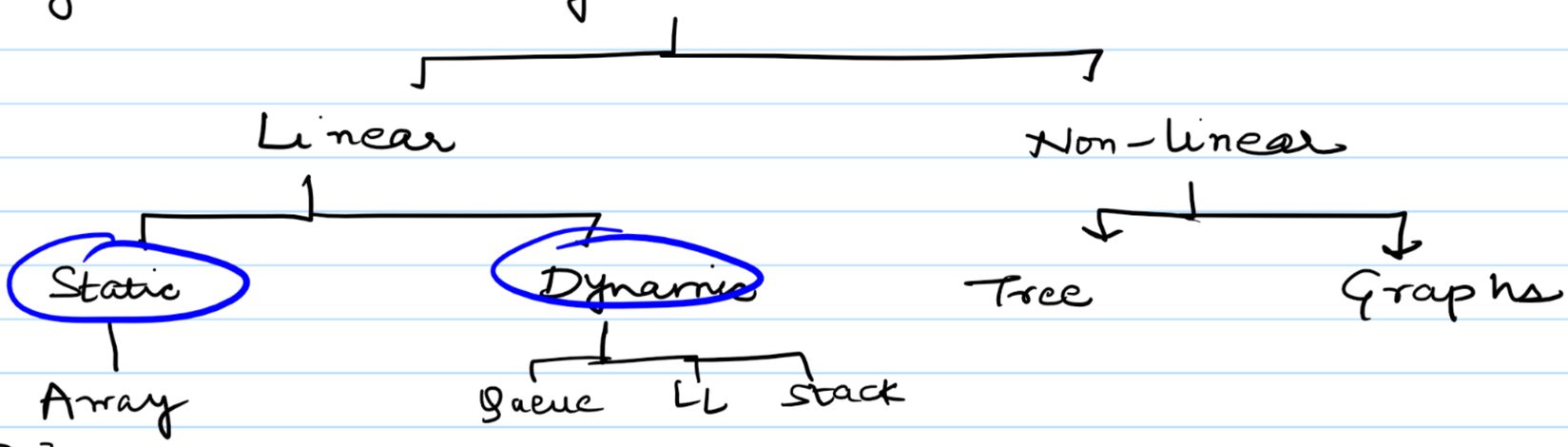
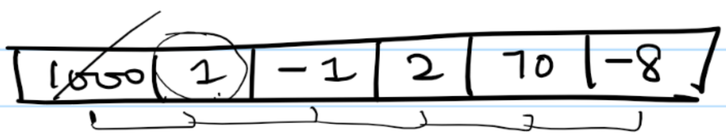
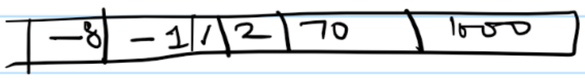
Data Structure

Storage used to organise data.





Brute force:



$n-1 + n-2 + \dots + 1 = \frac{n(n+1)}{2}$

$= O(n^2)$ $= \frac{n^2+n}{2}$

2nd solution:

1) Sort the array
2) return 1st (min) & last element (max)

$O(n \log n)$

3rd solution:

1) Initialise min & max by 1st element of the array
2) Keep on comparing the elements on the go
3) Return the min & max

		-1	8	3	2	-5
Max	0	0	8	8	8	8
Min	0	-1	-1	-1	-1	-5

Max element = 8
Min element = -5

TC : $O(n)$

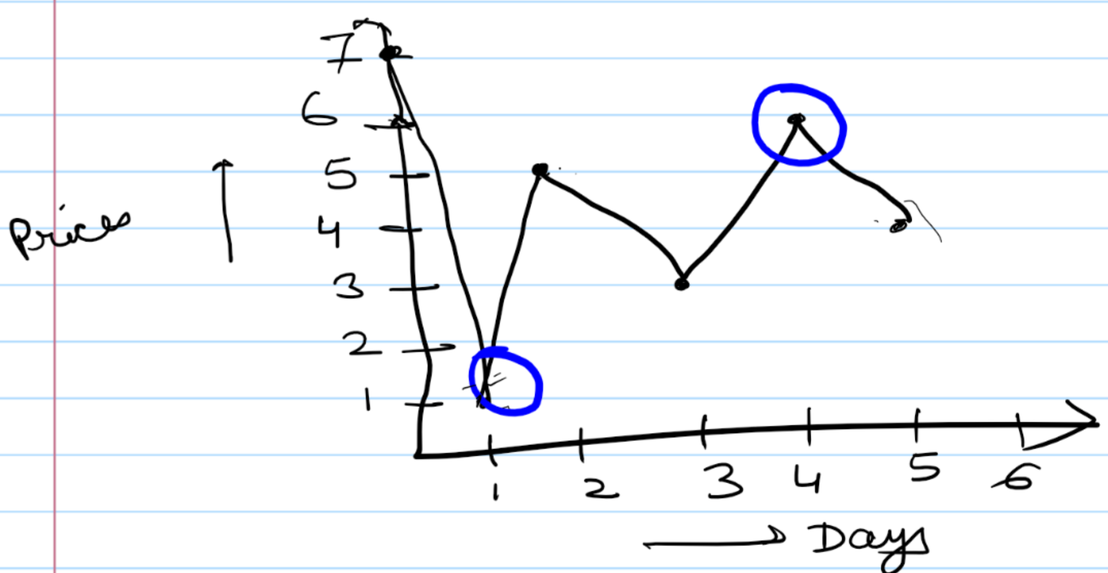
Q2 Best time to buy and sell stock

[7, 1, 5, 3, 6, 4]
0 1 2 3 4 5

Max = 7 Min = 1 = Profit = -6

Brute Force: 7 1 5 3 6 4

$n-1 + n-2 + \cdots + 1 = O(n^2)$



6 - 1 = 5

Day	0	1	2	3	4	5
Profit =	0	0	4	2	5	3
Min =	7	1	1	1	1	1
Maxprofit =	0	0	4	4	5	5

→ Return 5 as max profit

Algo

1) let max_profit = 0

2) For each position, calculate the min_value till now.
   if min_value > price
   min_value = price
   else
   Calculate profit = price - (min_value)

3) Max_profit = max(max_profit, profit)

TC: O(n)
SC: O(1)

9 Maximum Product subarray

[2, 3, -2, 4]

Brute Force: 6, -12 → $n-1 + n-2 + \cdots + 1 = O(n^2)$

① Zero = 0

② -ve numbers = -ve / +ve

2 3 -2 -4 5

2 6 -12 48

Max_till_now

Min_till_now

1 -2 7

Max →	1	-2	-14	(7)
Min →	1	-2	-14	
curr →	1	-2	7	

Result = 1 1 7

Max_till_now = Max (max_till_now * curr,
min_till_now * curr,
curr)

Min_till_now = Min ( )

	1	-2	7	
Max_till_now	1	-2	7	
Min_till_now	1	-2	-14	
curr	1	-2	7	Return 7
Result	1	1	7 →	

Max = max (1 * -2, 1 * -2, -2)

Max = max (-2 * 7, -2 * 7, 7)

Algo: Initialise result = 0

1) While going through numbers, we have to keep track of maximum product up till that no (max_so_far) and min " " (min_so_far)

2) Max_so_far = max (max_so_far * curr, min_so_far * curr, curr)

Min_so_far = min ( )

3) Update result = Max (result, max_so_far)

Q4: [-1, 0, 1, 2, -1, 4]

3 sum problem

Brute force: $O(n^3)$

2 - sum problem

-5 -4 2 3 9 sum = 5

2 pointer

-5 + 9 = 4 -4 + 9 = 5 2 + 3 = 5

2 + 9 = 11

Ans: [-4, 9], [2, 3]

3 sum ⟶ 2 sum

$-\underline{1} + x + y = 0$

$x + y = 1$

Algo:

1) Sort the input array

2) → If current value > 0, break from the loop

→ If current value is same as one before, skip it

→ call 2sum for current position (i)

2 sum

I) Set lo = i+1, hi = last index

2) while lo < hi

→ if sum of num[lo] + num[hi] < -num[i]

or

num[lo] + num[hi] + num[i] < 0

increment lo

→ if > 0 decrement hi

otherwise = 0 add triplet num[i], num[lo]

& num[hi] to ans

increment lo, decrement hi pointer

TC: sorting = O(nlogn)

n * (2 sum)

n * (n) = n^2

= $n^2 + n\log n$

= $O(n^2)$

2, 2, 2, 3, 3 — 5

{2,3}, ~~[2,3] [2,3]~~ = [2,3]

Q K^{th} largest element in an array

[3, 2, 1, 5, 6, 4]

Brute force: Sort the array
Return $n-(K-1)^{th}$ element from the end

O(nlogn)

1 2 3 4 (5) 6

2nd largest

Hint : Priority Queue Heap

complete binary tree

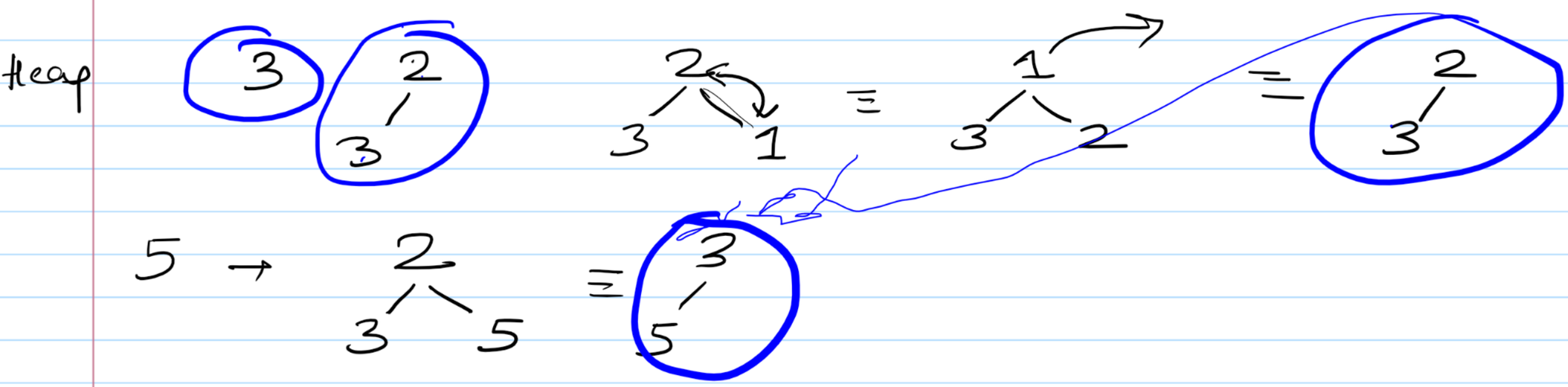
Max-heap (parent node value > its children value)

Min heap (P < C)

3, 2, 1, 5, 6, 4 K = 2

3, 2, 1

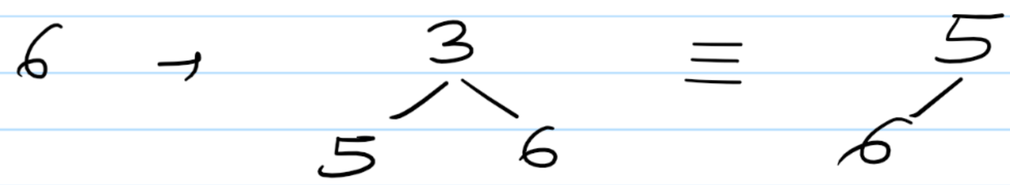
Heap



5 →

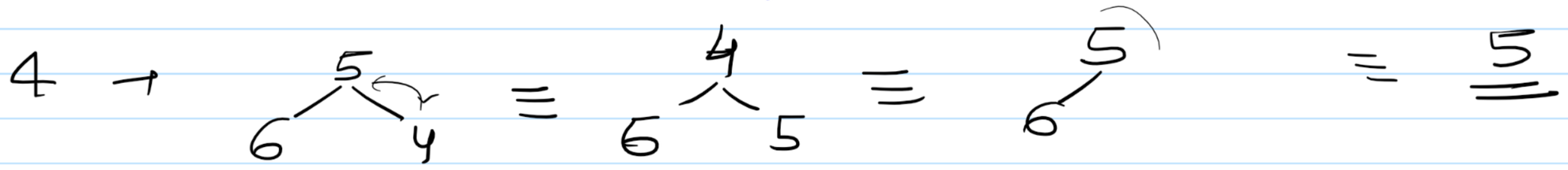
3, 2, 1, 5

6 →



3, 2, 1, 5, 6

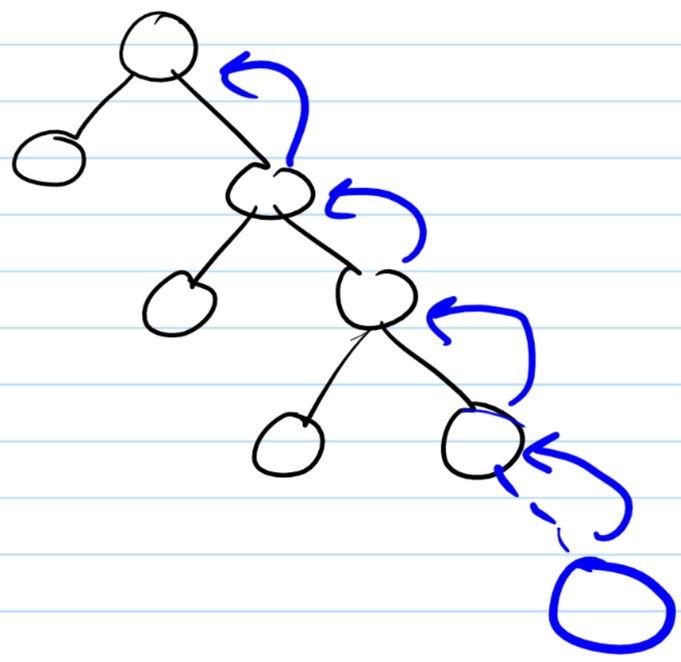
4 →



Algo 1) create a min heap & add all elements

from the array into the heap 1 by 1

2) Heap will store k largest element at any point
Head of this heap will be the answer



k size heap
height = O(log K)
n elements

$\boxed{n * O(\log k)}$ ← TC

k size heap = O(K) ← SC