Problem Solving Session on Heaps - 2

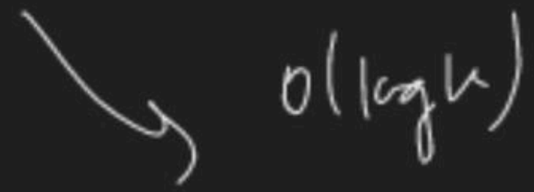
Special class

Rohit Mazumder • Jan 3, 2021

When to use the heap DS?

The heap data structure is to be used when we have to perform the operations dynamically:

1. insert(value) → O(log N)
2. getMin() or getMax() → O(1)
3. extractMin() or extractMax() → O(log N)



Sort the array!

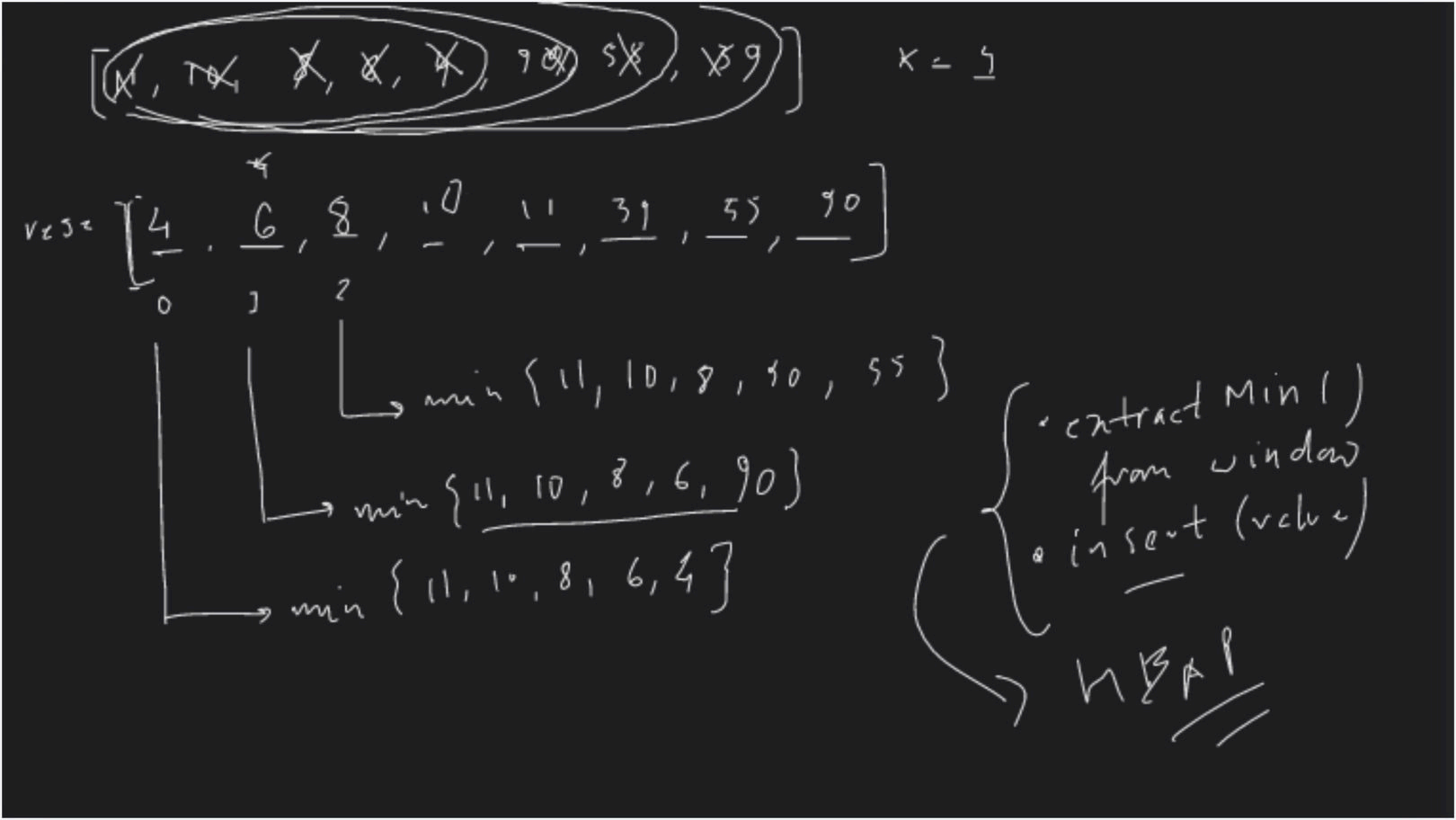
You have been provided with an array **A** of length **N**. Your task is simple - sort this array!
How efficiently can you sort this array if it is guaranteed that each element of this array is at-most at a distance of **K** from its sorted position?
Input - N, K and A

Example:
N = 8, K = 4
[11, 10, 8, 6, 4, 90, 55, 39]
Output:
[4, 6, 8, 10, 11, 39, 55, 90]

distance from sorted position <= 4 = K

O(N log N)

(K+1)


k = 3
vec = [4, 6, 8, 10, 11, 31, 55, 90]
0 1 2
min {11, 10, 8, 90, 55}
min {11, 10, 8, 6, 90}
min {11, 10, 8, 6, 4}
• extract Min ()
from window
• insert (value)
HEAP

An element at index i, will be placed somewhere in **[max(0, i - K), min(i + K, N - 1)]** in the sorted array.

For K = 4,

[11, 10, 8, 6, 4, 90, 55, 39]

11 will be present somewhere in [0: 4] in the sorted array.

[11, 10, 8, 6, 4, 90, 55, 39]

90 will be present somewhere in [1: 7] in the sorted array.

Let's integrate a heap to our algo!

1. Push the first K + 1 elements from the array to the min-heap. $O(K \log K)$ insert()
2. Fill the res array: $\longrightarrow O(N)$
   a. Extract the minimum element is the res array. $\longrightarrow O(\log K)$
   b. Put the next element from the array into our min-heap (if present). $\longrightarrow O(\log K)$ insert()

$O(K \log K + N \log K) \approx O(N \log K)$ (K < N)

$O(N \log N)$

[11, 10, 8, 6, 4, 90, 55, 39] k = 4

[4, 6, 8, 10, 11, 39, 55, 90]

O(N)

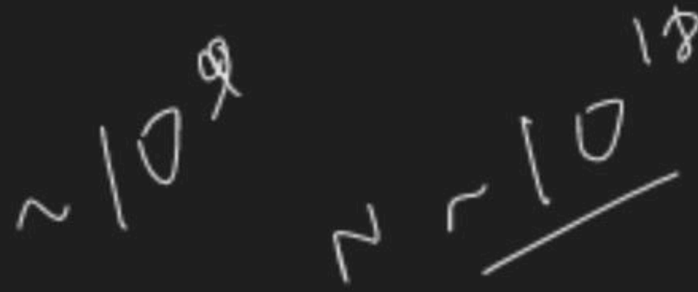
min-heap

Kth largest in the stream

Find the **K**th largest element in the stream of N integer. (K < N)

Input:
First line contains two spaced integers - N and K
The next N lines contains a single integer - the element of the stream.

Output:
The Kth largest element in the stream so far. If the stream has less than K elements then output "#"



Example:

Input:

```
7 3
33
10
25
178
3
192
200
```

Output:

```
#
#
10
25
25
33
178
```

N = 7 ; K = 3 3RD largest number.

[33] → #

[33, 10] → #

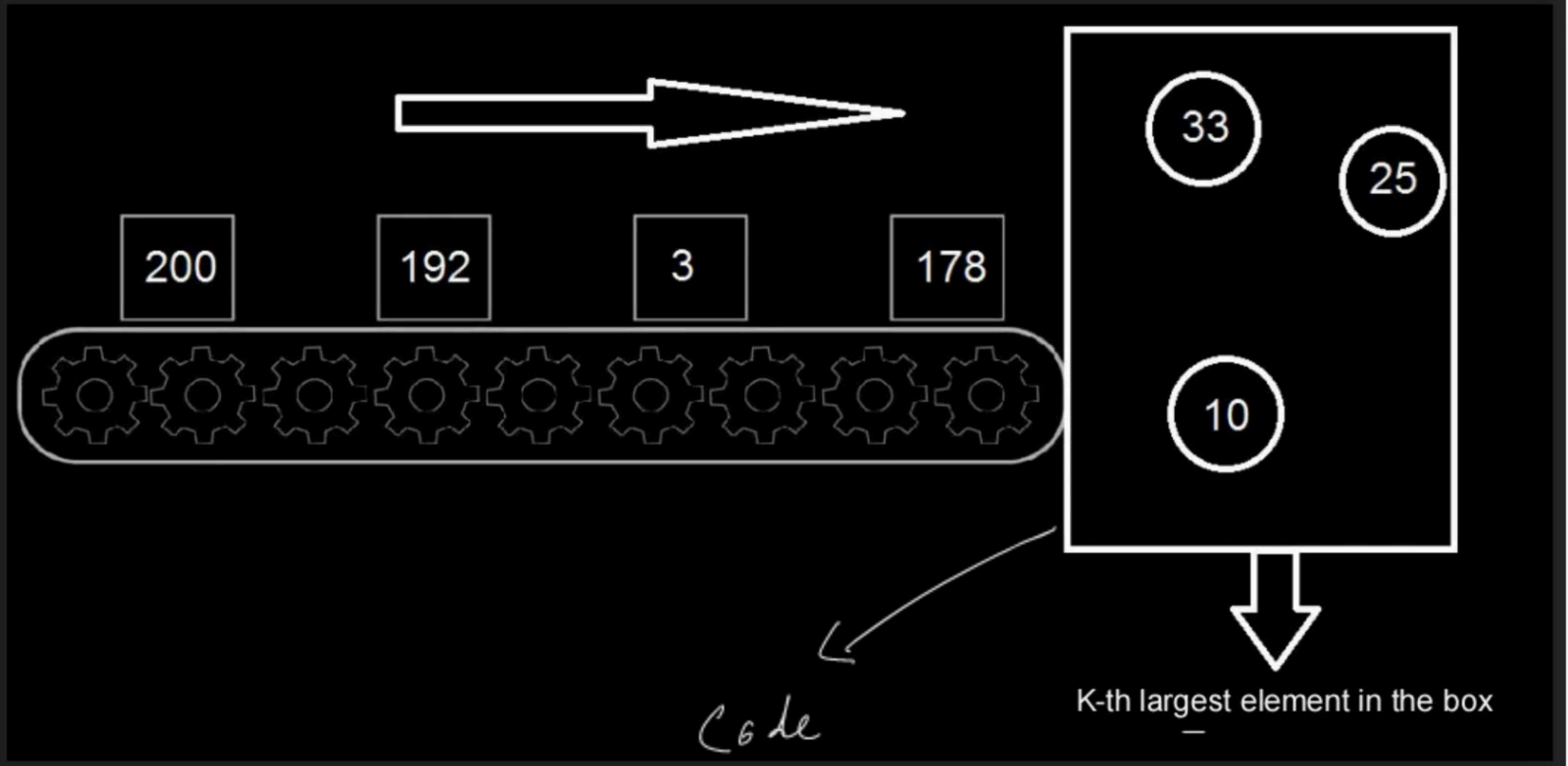
[33, 10, 25] → 10

[33, 10, 25, 178] → 25

[33, 10, 25, 178, 3] → 25

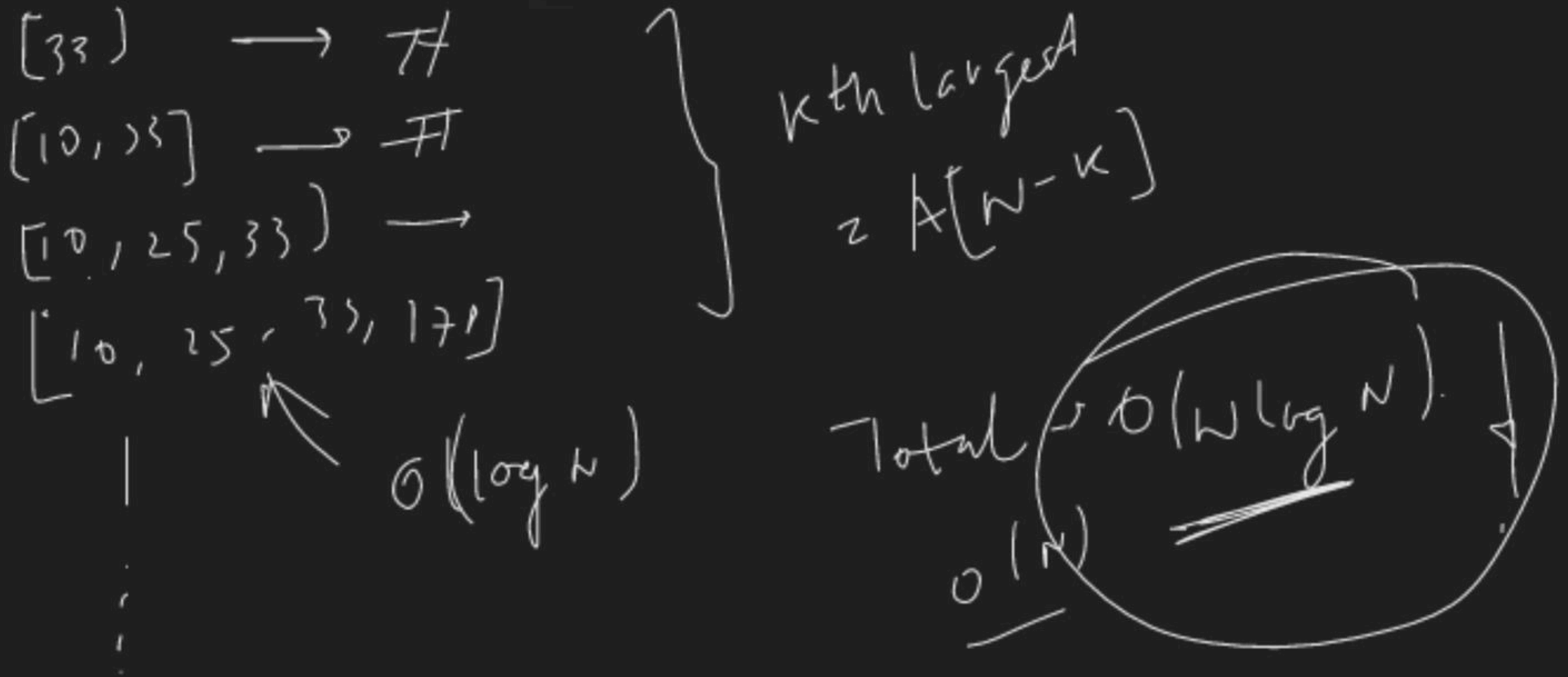
[33, 10, 25, 178, 3, 192] → 25

[33, 10, 25, 178, 3, 192, 200] → 178


200
192
3
178
33
25
10
K-th largest element in the box
Code

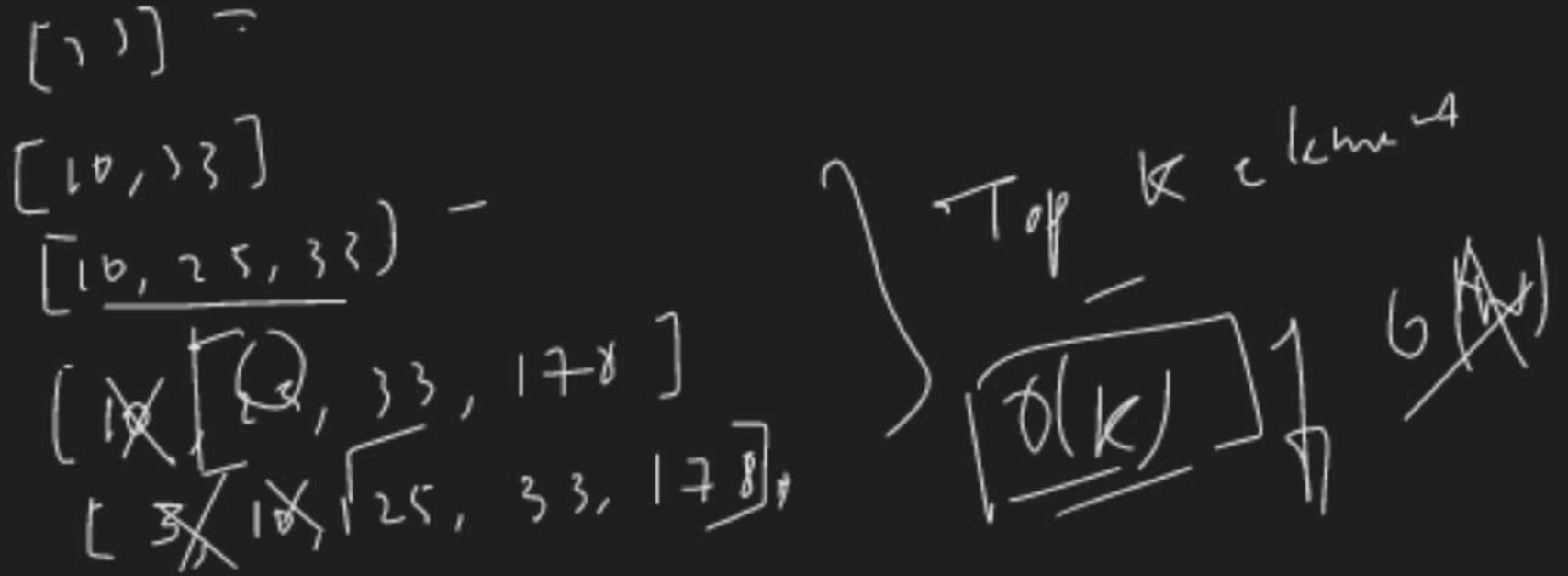
Possible Solution-1:

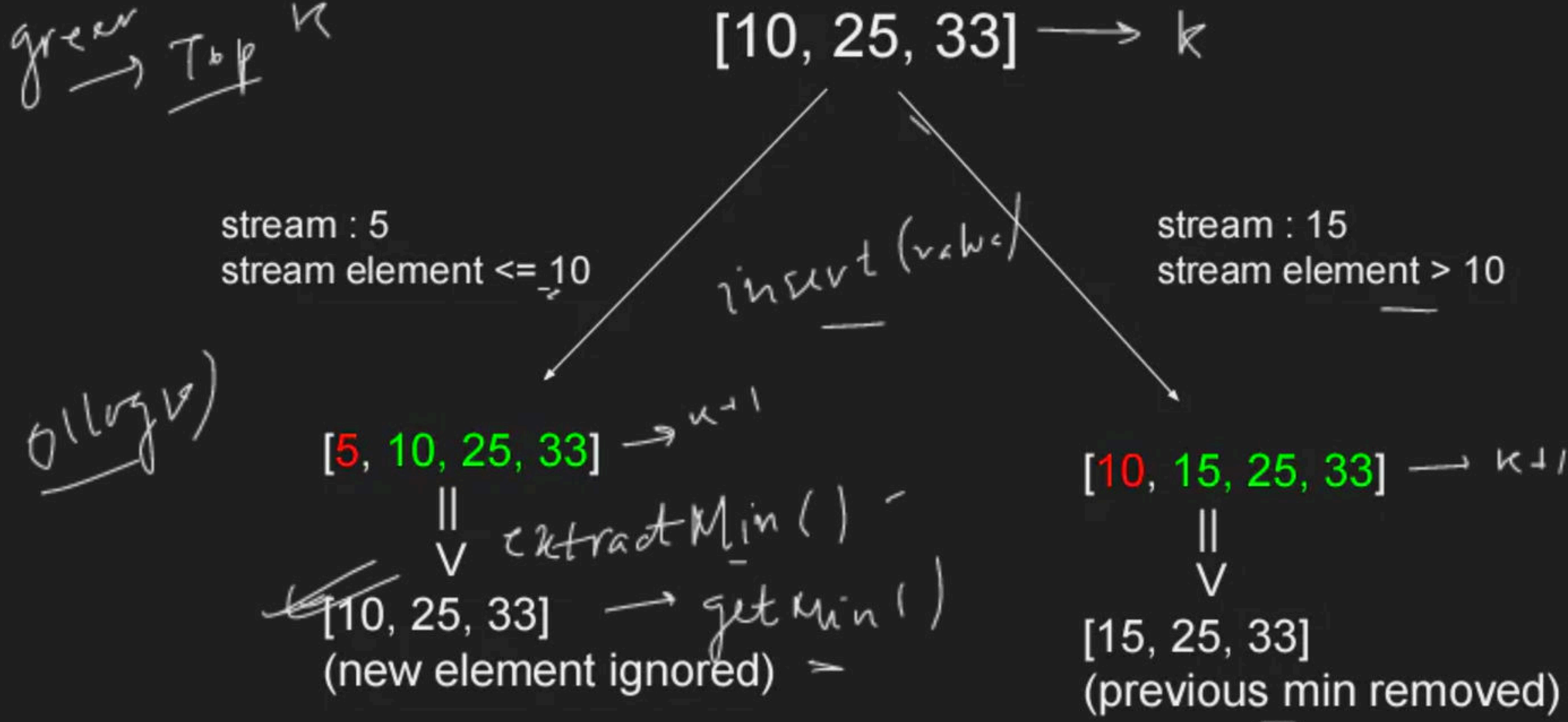
Maintain a sorted list of integers.



Do we need to actually keep track of all N elements? What if we just keep track of the top K elements?

If we can maintain the top K elements, the Kth largest element of the stream will the minimum element among the top K elements.





Combining these two cases, we can say that we everytime we get the new stream element, we can add it to our list and then extract the minimum of the K + 1 elements to retain the top K elements! And once you have the top K elements, you can get the minimum value from them!

Integrating the heap to our algo:

1. Add the first K elements to the heap and output "#" for each of them. $O(K \log K)$
2. For the rest of elements do the following: $\longrightarrow O(N)$
   a. insert(newStreamValue) //Add the new value to the heap $\longrightarrow O(\log K)$
   b. extractMin() //Extract the min to retain the top K values. $\longrightarrow O(\log K)$
   c. Output getMin(). $\longrightarrow O(1)$

$O(K \log K + N \log K) \sim \boxed{O(N \log K)}$ ✓

$K << N$

Output

#

#

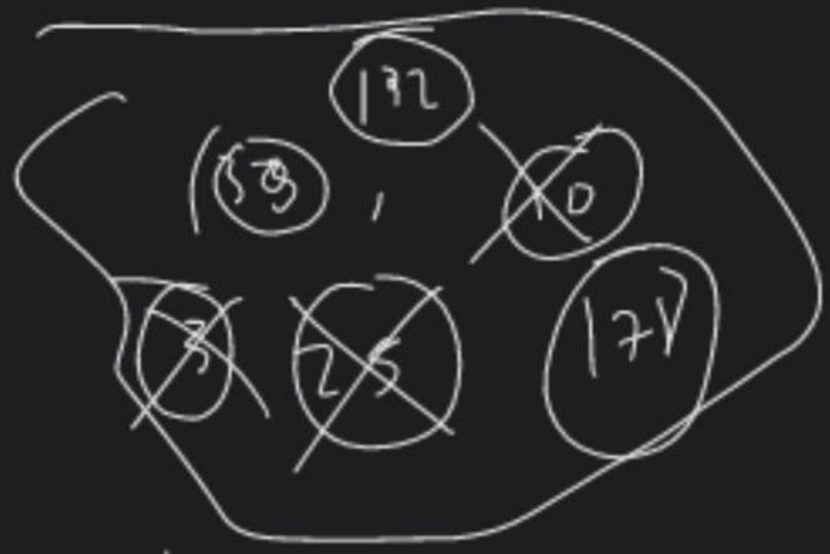
10

25

.

.

.



min-Heap

RRATING

You have to perform Q queries on a stream of integers. Each query is of one of the two types -

1 x - Add element x to your stream.
2 - Output (N/3)th largest element. In case the stream contains less than 3 elements output "#"

[NOTE: While calculating N/3, take the floor value]

$\lfloor \frac{N}{3} \rfloor$

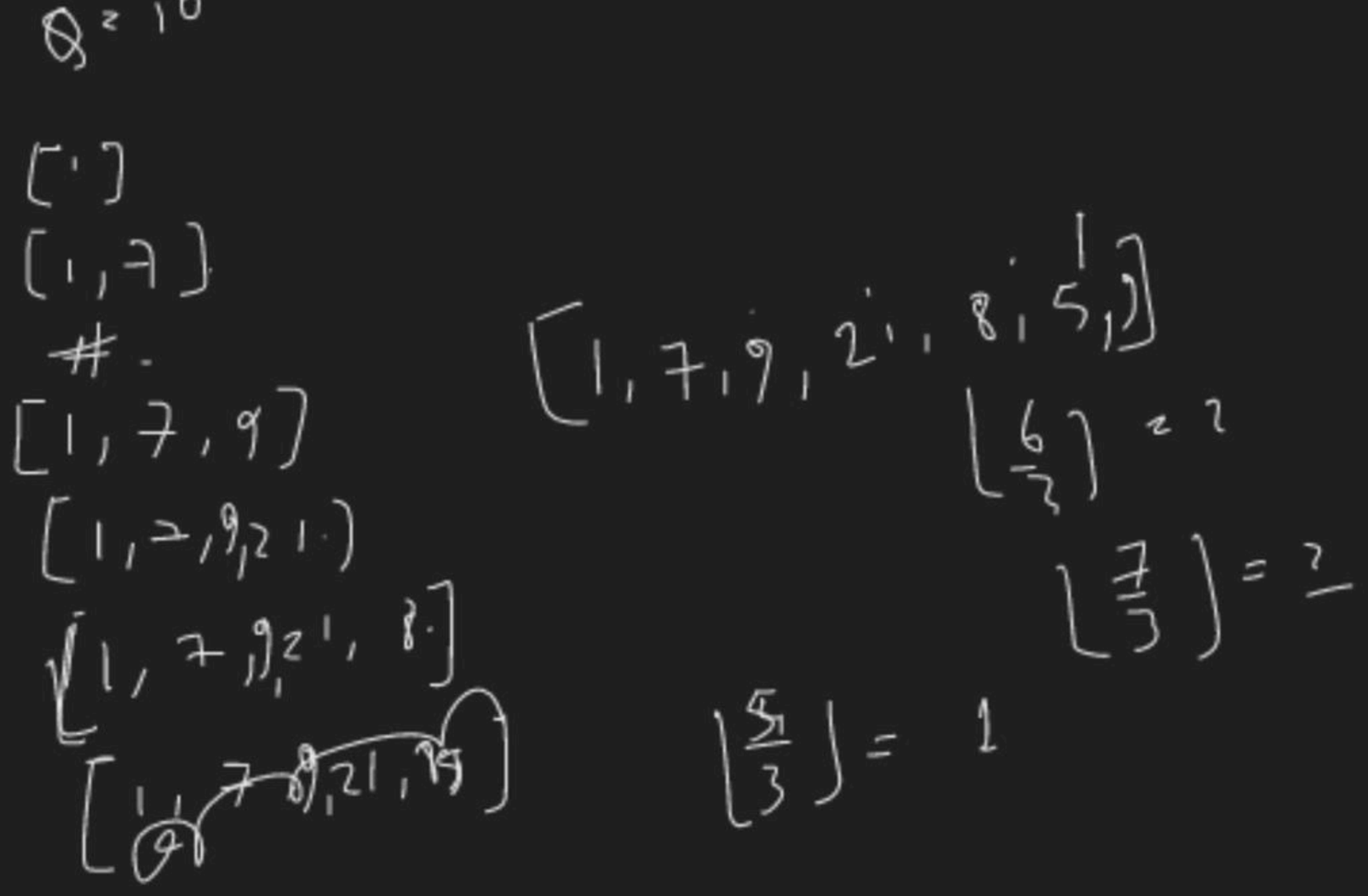
Example

Input:

```
10
11
17
2
19
121
18
15
2
19
2
```

Output:

```
#
9
9
```



Case 1: When input <= max element in 2/3rd section

Stream; Red -> 2/3rd section, Green -> 1/3rd section.	N	A[N/3] ↓ Query - 2
A: [1, 2, 3, 4, 5, 6]	6	5
A: [1, 2, 3, 4, 4, 5, 6] ; insert = 4	7	5
A: [1, 2, 3, 4, 4, 4, 5, 6] ; insert = 4	8	5
A: [1, 2, 3, 4, 4, 4, 4, 5, 6] ; insert = 4	9	4

$\lfloor \frac{7}{3} \rfloor = 2$

$\lfloor \frac{8}{3} \rfloor = 2$

$\lfloor \frac{2}{3} \rfloor = 3$

Case 2: When input > max element in 2/3rd section

Stream; Red -> 2/3rd section, Green -> 1/3rd section.	N	A[N/3]
A: [1, 2, 3, 4, 5, 6]	6	5
A: [1, 2, 3, 4, 5, 6, 7] ; insert = 7	7	6
A: [1, 2, 3, 4, 5, 6, 7, 8] ; insert = 8	8	7
A: [1, 2, 3, 4, 5, 6, 7, 8, 9] ; insert = 9	9	7

$\lfloor \frac{7}{3} \rfloor = 2$

$\lfloor \frac{8}{3} \rfloor = 2$

$\lfloor \frac{9}{3} \rfloor = 3$

Conclusions:

1. When N%3 == 0; there's a change in the number of elements in the top 1/3rd section.
    - When input > max element in the 2/3rd section; input will be a part of the top 1/3rd.
    - When input <= max element in the 2/3rd section; the largest element from the 2/3rd section will become a part of the 1/3rd

2. When N%3 != 0;
    - When input > max element in the 2/3rd section; input will be added to 1/3rd section and the minimum element from 1/3rd section will be moved to 2/3rd section.
    - When input <= max element in the 2/3rd section; input will be added to the 2/3rd section

2. We need an efficient way to calculate the min element of 1/3rd section and maximum element of 2/3rd section.

min-heap — max-heap