→ Heap Sort

→ K^{th}-largest element

→ Sort nearly sorted array

→ median of stream of integers.

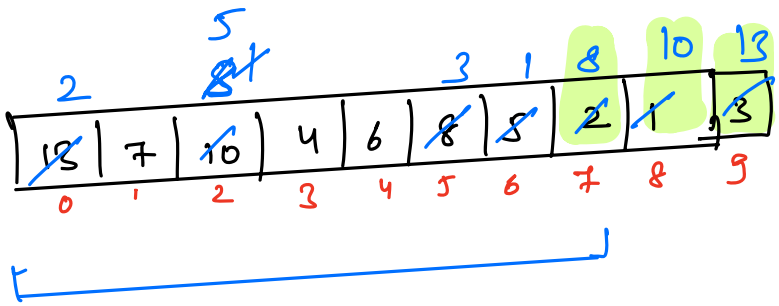
Sort the array

idea → Build the min-Heap and keep extracting/removing the minimum element & store it in ans[]

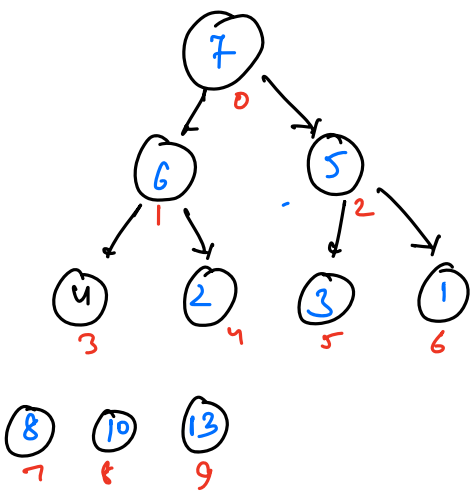
[T.C → O(NlogN), S.C → O(N)]

↓

Can we optimise the space?



max-Heap.



Heap Sort

① Build maxHeap

```
for( i = N/2 - 1 ; i >= 0 ; i--){
        heapify (arr[], i, N-1);
}
```

② j = N-1

```
while( j > 0){
        swap( arr[0] with arr[j]);
        j--;
        heapify( arr[], 0, j);
}
```

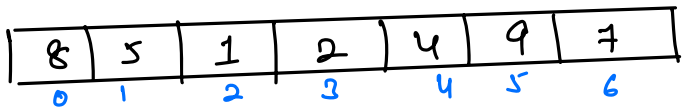
T.C → O(N log N)
S.C → O(1)

→ in place ✓

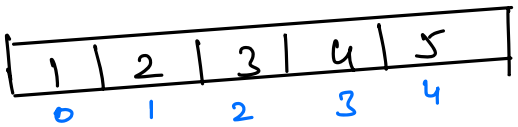
→ stable sorting algo? X

∵ Relative position of similar elements can change.

Q) Given arr[N]. Find k^{th} largest element.

arr[] → 

[k=3] ans = 7.

arr[] → 

[k=5] ans = 1.

idea-1. Sort the array & return arr[N-K]

T.C → O(NlogN), S.C → O(1)

idea-2. Insert all the elements in maxPQ / maxHeap

↓

ExtractMax() K-1 times

[T.C → O(NlogN + KlogN)
S.C → O(N)]

idea-3. Using minHeap

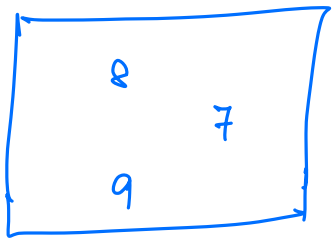
b1	b2	b3	b4	b5	b6	b7	b8
↓	↓	↓	↓	↓	↓	↓	↓
12	8	4	6	7	3	10	9

12, 8, 10, 9

arr[] →

8	5	1	2	4	9	7
0	1	2	3	4	5	6

[K = 3]



minPQ

K largest elements are now present in minHeap / minHeap.

ans = 7.

#code →

① Insert first K elements in minPQ / minHeap.

② for(i = K ; i < N ; i++){

    if(arr[i] > minPQ.getMin()){

        minPQ.removeMin();

        minPQ.insert(arr[i]);

    }

}

③ ans → minPQ.getMin();

$K \log K + (N-K) \log K$

$K \log K + N \log K - K \log K$

T.C → O(N log K)

S.C → O(K)

K^th smallest element →

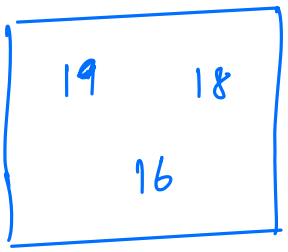
idea → use max. heap of size-K.

Q) k^{th} largest element for all the windows of $(0, i)$ $(i = k-1)$

arr[] - [10 18 7 5 16 19 3] , k=3

0 1 2 3 4 5 6

ans → [7, 7, 10, 16, 16] Expected output.



ans → [7, 7, 10, 16, 16]

code → todo

Q) Given a nearly sorted array. You need to sort the array.

⇓

Every element is shifted away from its correct position by atmost k-steps.

13	22	31	45	11	20	48	60	50
0	1	2	3	4	5	6	7	8

k=4

minimum element can be present from idx=0 to idx=k

⇒ Take minHeap / minPQ of size k+1.

~~48~~, ~~50~~ ~~45~~ ~~31~~, ~~60~~

[11, 13, 20, 22, 31, 45, 48, 50, 60]

code →

① Insert first k+1 elements in minHeap / minPQ.] k log k

② for(i = k+1; i < N; i++){

minPQ.removeMin() → ans[]

minPQ.insert(arr[i])

}] (N-k) log k

③ while(minPQ.size() > 0){

minPQ.removeMin() → ans[]

}] k log k

④ return ans[]

[T.C → O((N+k) log k)
S.C → O(k)]

Q) Given an infinite stream of integers. Find the median of current set of elements.

median ⇓ middle element in sorted array

6	[6] →	6
6, 3	[3, 6] →	4.5
6, 3, 8	[3, 6, 8] →	6
6, 3, 8, 11	[3, 6, 8, 11] →	7
6, 3, 8, 11, 2	[2, 3, 6, 8, 11] →	6

B.f. idea.

For every incoming element.

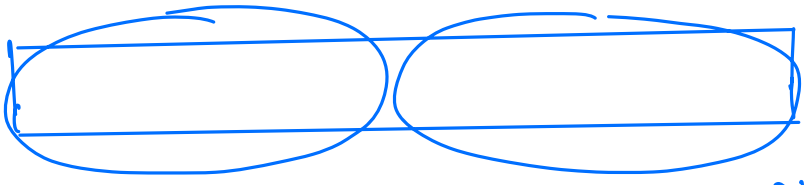
sort the array and find the middle or avg. of two middle elements.

T.C → $O(N^2 \log N)$

idea-2. Every time, find the correct position of the upcoming element i.e. Insertion Sort.

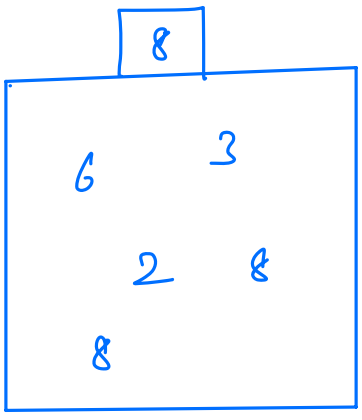
T.C → $O(N^2)$

idea-3.

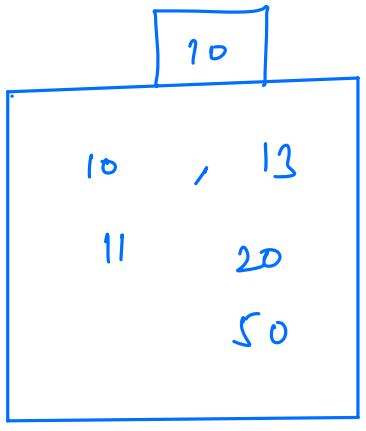


$\frac{N}{2}$ smaller elements | larger elements $N/2$

arr- [6, 3, 8, 11, 20, 2, 10, 8, 13, 50, — —]



max-heap (h1)



min-heap (h2)

$|h1.size - h2.size| \leq 1$

median:- [6, 4.5, 6, 7, 8, 7, 8, 8, 8, 9, — —]

code →

h1, h2 — h1.insert(arr[0]), print(arr[0]);

max-heap min-heap

```
for( i = 1;  i < N;  i++){
    if( arr[i] <= h1.getMax()){
        h1.insert(arr[i]);
    }
    else{
        h2.insert(arr[i]);
    }
    diff = Abs| h1.size() - h2.size()| ;
    if( diff > 1){
        balance(h1, h2);
    }
    if( h1.size() > h2.size()){
        print( h1.getMax());
    }
    else if( h2.size() > h1.size()){
        print( h2.getMin());
    }
    else{
        print((h1.getMax() + h2.getMin()) / 2.0);
    }
}
```

[T.C → O(N log N)
S.C → O(N)]

```
void balance ( h1, h2) {
    if ( h1.size() > h2.size()) {
        h2.insert ( h1.removeMax());
    }
    else {
        h1.insert ( h2.removeMin());
    }
}
```

Greedy → Sorting, comparator
       → P.Q.

minPQ

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

```
PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
```

maxPQ