

# Heaps 2

[Heap Sort](#)

[Kth Largest Element](#)

[Kth Largest Element in every prefix](#)

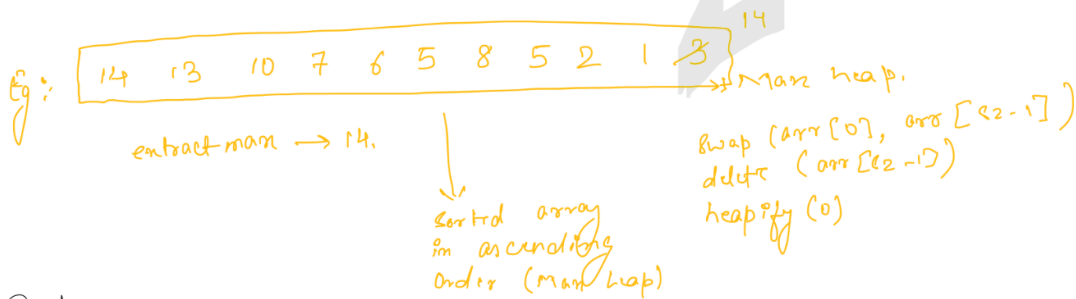
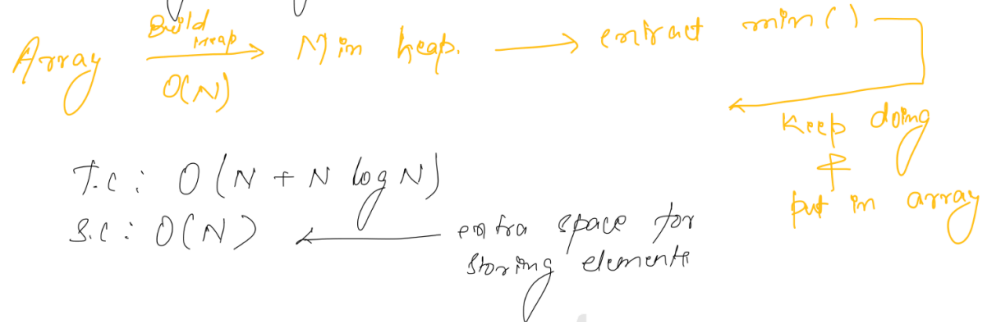
[K sorted Array](#)

[Running Median](#)

# Heap Sort

Heap Sort is

→ Sort an array using heap. (increasing).



Pseudo code

1) Build Max heap  $O(N)$

2)  $j = N-1$

```

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

# Kth Largest Element

Ex: Find min  
 - Ex: 

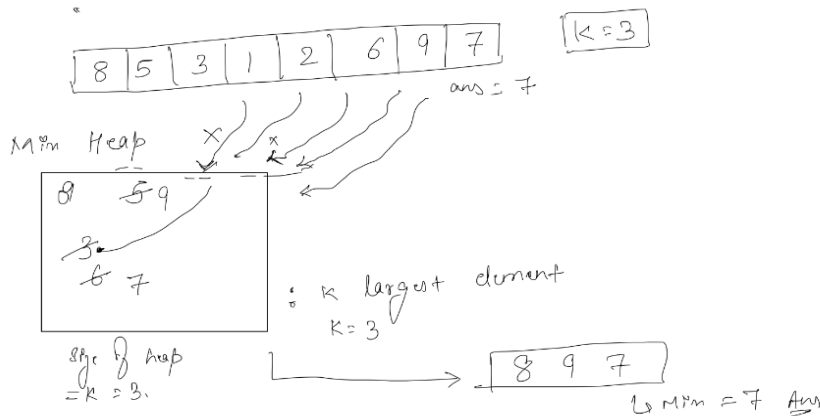
8	5	3	1	2	6	9	7
---	---	---	---	---	---	---	---

K=3  
 ans = 7

Idea I → Sort the array in descending order & get kth index.  
 ↳ T.C:  $O(N \log N)$

Idea II Use max heap :-  
 Build max heap of the array }  $\log N$   
 ↓  
 extractMax() K times }  $K * \log N$       T.C:  $O(N + K \log N)$

Idea III Min heap :-



Pseudo Code :-

```

1) Insert first K elements simply into a min heap
   for (int i=0; i<K; i++)
   {
       insert(heap, arr[i])
   }

2) for (int i=K; i<N; i++)
   {
       if (arr[i] > getMin(heap))
       {
           extractMin(heap);
           insert(heap, arr[i]);
       }
   }

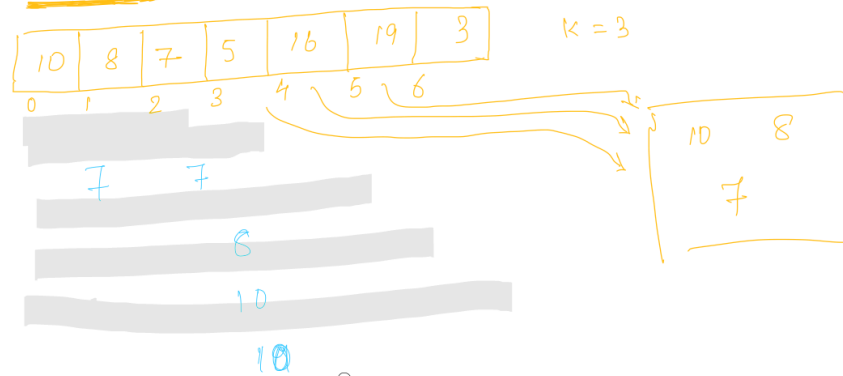
3) ans = getMin();
    
```

T.C:  $O(K \log K + (N-K) \log K)$

T.C:  $O(N \log K)$

# Kth Largest Element in every prefix

Q  $k^{\text{th}}$  largest element for every window  $(0-i)$   
 where  $i \geq k-1$



→ Maintain a min heap of size  $k$ .

→ Insert first  $k$  elements in heap.

exactly same as previous

(Modify heapify with  
 $\text{heapify}(\text{hp}[i], i, k)$ )

# K sorted Array

## K sorted Array

Every element is at max  $k$ -distance away from its sorted position. Sort the array as



Idea 1 is Sort the array  
T.C:  $O(N \log N)$

Idea 2 is: 0th element  $\longrightarrow$  Smallest element  $\longrightarrow [0-k]$   
Min heap  $[k]$

1) Insert all elements from  $0-k$  in a min heap.

2)  $arr[0] = \text{extract Min}()$  // get and remove.

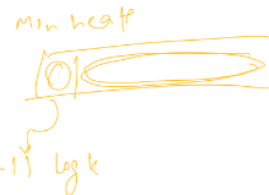
// first element  $\downarrow$   
for (int  $i = k+1$ ;  $i < N$ ;  $i++$ )

{  
     $mh.insert(arr[i])$   
     $arr[i] = \text{extract Min}()$ ;  
     $i++$ ;  
}

// element from  $1-N-k$

while (mh is not empty)

{  
     $arr[i] = mh.extract Min()$ ;  
     $i++$ ;  
}



$k \log k$

T.C:  $N \log k$

# Running Median

## Q4 Running Median :-

Running stream of integers. Find the median of the element after every new element joins.

Median :- Middle element in sorted Array.

arr[3] : { 1, 4, 2 }  $\rightarrow 2$

arr[6] : { 1, 2, 1, 4, 3, 6 }  $\rightarrow 3, 4$   
 $\text{avg} : 3.5 = 3$

Brute force

1) After every new element joins, sort the array.

$$N \times (N \log N)$$

$$\boxed{T.C : N^2 \log N}$$

Optimized Approach

Eg:- { 3, 7, 11, 5, 13, 23, 20, 17, 25 }  $N=9 \rightarrow \text{odd}$   
 $\swarrow \searrow$   
 5 4

3	7	11
5	13	

Left bucket

$\leq$

23	20	17
25		

Right bucket

$\boxed{\text{Median} = \text{Max Element of left.}}$

Eg:- { 3, 8, 7, 4, 10, 14, 29, 24, 30, 32 }  $N=10$   
 even

$\swarrow \searrow$   
 $N/2$

3	8	7
4	10	

Left bucket

$\leq$

14	29	30
24	32	

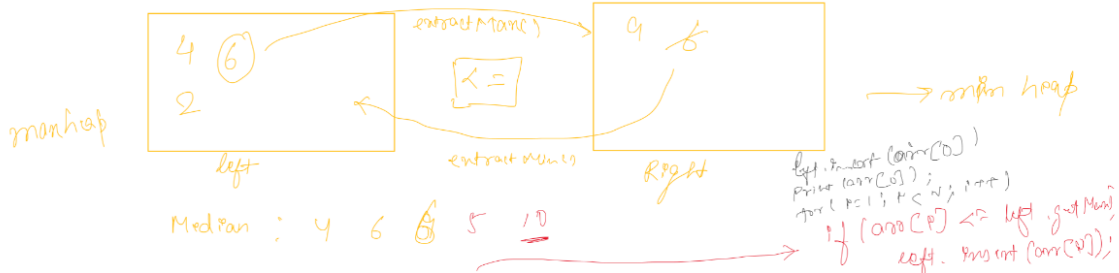
Right bucket

$\boxed{\text{Median} = \frac{\text{Max on left} + \text{Min on Right}}{2}}$

Operation:-

arr[]: { 4, 9, 6, 2, 1, 7, 0, 9, 7, 3, 5 }

6 > max of left bucket



```

if (left.size() < right.size())
{
    left.insert(right.extractMax());
}
else if (left.size() > right.size() > 1)
{
    right.insert(left.extractMax());
}
// Check for even & odd
if (left.size() == right.size())
{
    // (left.getMax() + right.getMax()) / 2
}
else {
    // left.getMax();
}

```

for integers  
streamline that  
works better :)

T.C:  $N \log N$

Use Built-in  
Priority Queue