Today's Content

- Heap Sort
- K^{th} Largest Element
- Sort Nearly Sorted Array
- Median of Stream of Integers

1. Sort the array using heap

arr: [3 1 9 4 6 2 10 5]

Idea 1:

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

1. Build a min heap
2. Extract Min, extract Min, extract Min
   - ↓ 1st min
   - ↓ 2nd min
   - ↓ 3rd min

↓

Extract min from heap 1 by 1 and put them in ans[] (sorted)

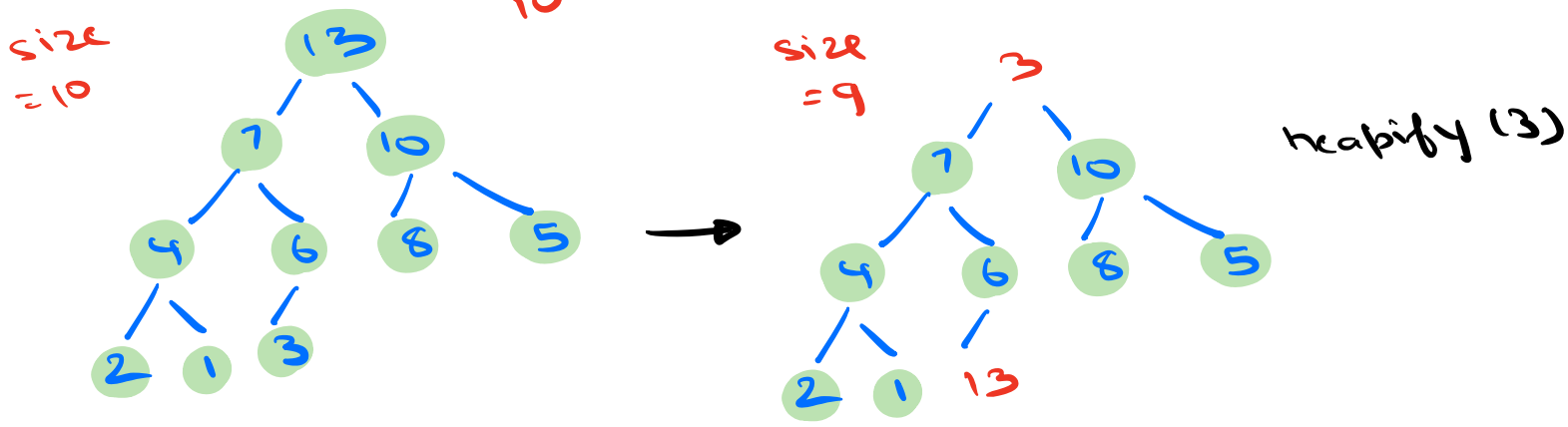
TC: O(N log N)

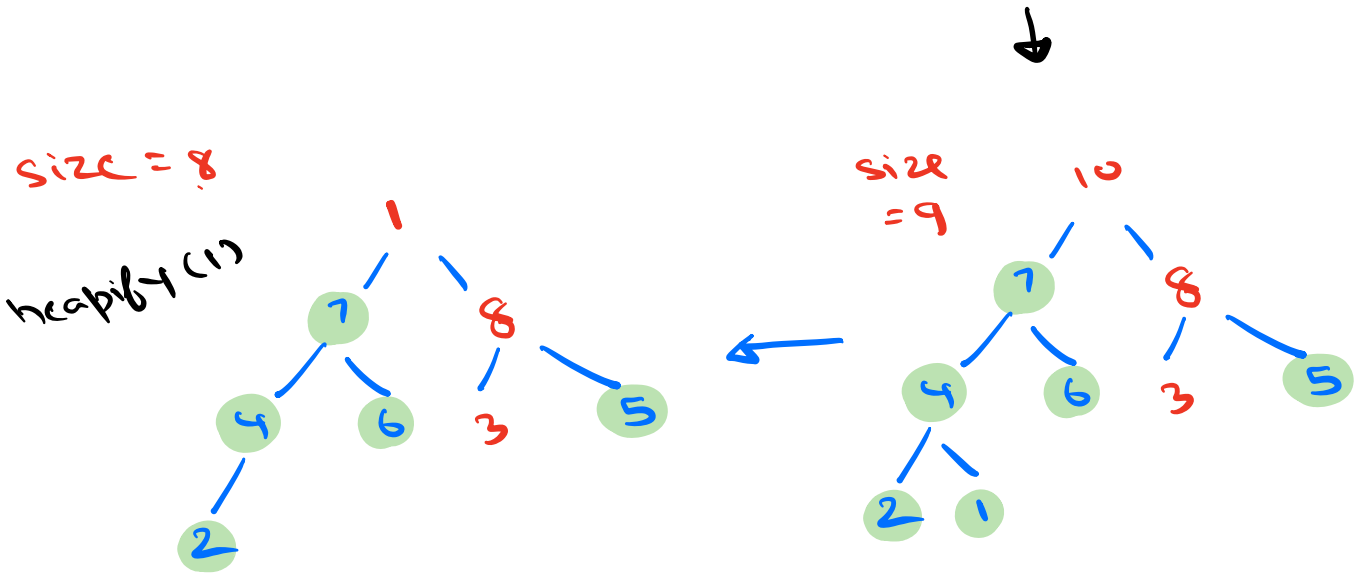
SC: O(N)

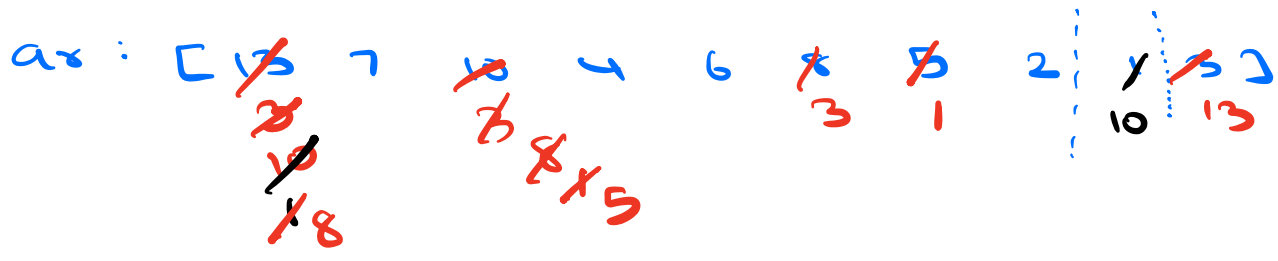
Idea 2: Can we optimize space?

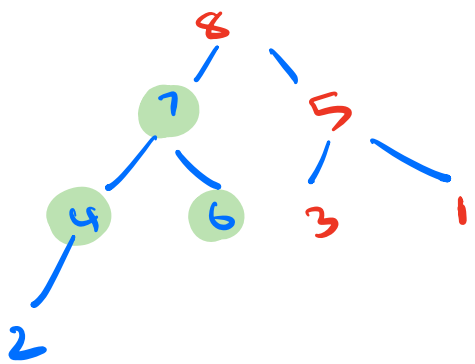
ans [ ]

arr: [13 7 10 4 6 3 5 2 1 13]









Heap Sort

Heap
↓
- Inplace sorting algo
- Not stable

TC : O(N lg N)
SC : O(1)

1. Build Max Heap
2. size = heap.size()
   while (size > 0) {
   swap (ar[0], ar[size-1])
   size--
   heapify (ar, 0, size)
   }

2. K^{th} Largest Element

ar : [8 5 1 2 4 9 7] k = 3

ar : [5 3 1 4 2] k = 5

Idea 1 : Sort data in descending order

↓

ar [k-1]

TC : O(N log N)

SC : O(N)

↓

Depends on sorting algo

OR

Sort in ascending order

↓

ar [N-k]

Idea 2 : Insert all de in max heap,
extractMax() k-1 times
target de (k^{th} largest) at root

TC : O(N + K log N) SC : O(N)

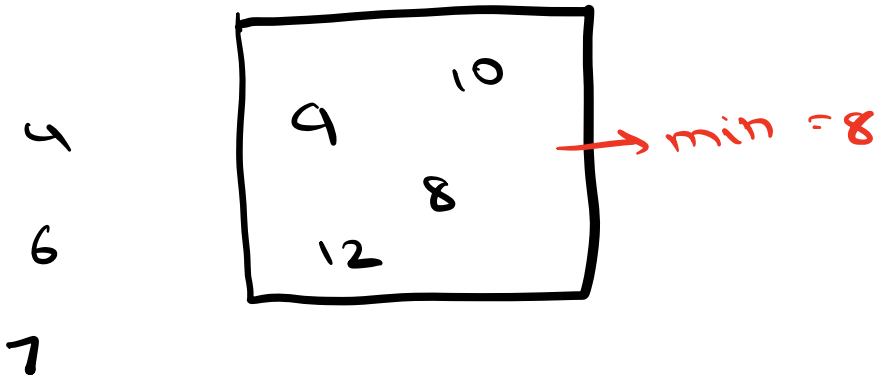
↓

modification not allowed in original array

Idea 3 : Min Heap

k=4
ans=8

[12 8 4 6 7 3 10 9]



1. Insert first k elements into minheap → i = 0 to k-1

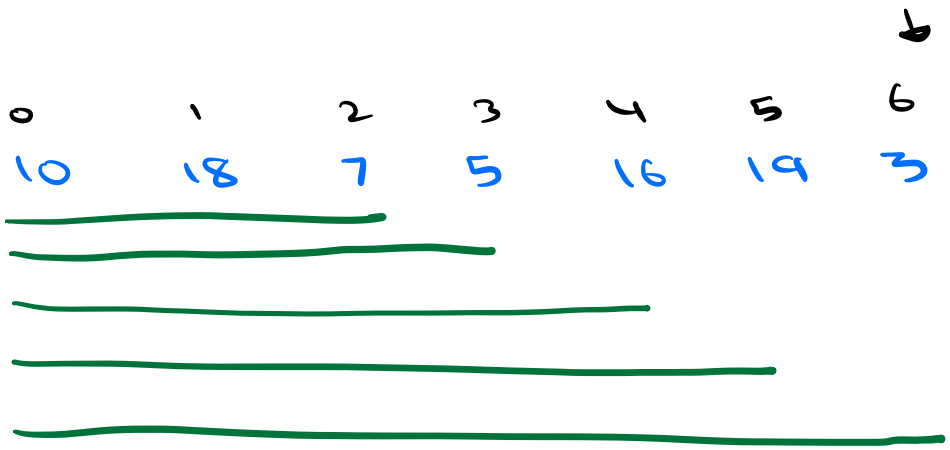
2. for (i = k ; i < n ; i++) {
    if (arr[i] > mH.getMin()) {
        mH.extractMin()
        mH.add(arr[i])
    }
}

return mh.getMin()

// Largest k el in MinHeap
// Top → kth largest

TC : O(k + (N-k)log k)
(k + Nlog k - klog k)
TC : O(Nlog k)
SC : O(k)

3. Kth Largest Element in all prefix subarrays (starting at index 0)



0-2 0-3 0-4 0-5 0-6

ans: [7, 7, 10, 16, 16]

idx	ans
0-2	7
3	7
4	10
5	16
6	16

[0 → k-1]

1. Insert 1st k elements into minHeap → O(k)
2. print (mH.getMin())
3. for (i=k ; i<n ; i++) { → (N-k) log k

```
    if (arr[i] > mH.getMin()) {
        mH.extractMin()
        mH.add(arr[i])
    }
    print (mH.getMin())
}
```

10:26

TC: O(k + N log k - k log k)

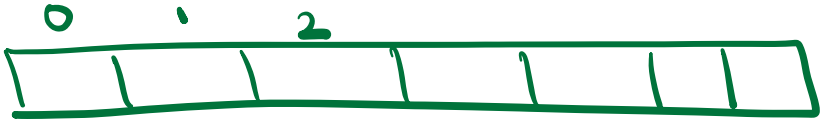
TC: O(N log k) SC: O(k)

4. The array is nearly sorted, sort it completely. Every element is shifted away from its correct position by atmost K steps.

	0	1	2	3	4	5	6	7	8	
ar →	13	22	31	45	11	20	48	60	50	K = 4
⇒	11	13	20	22	31	45	48	50	60	

Idea 1: Sort the entire array TC: O(N log N)

Idea 2:



idx	idx in orig[]
0	0 → K
1	0 → K+1
2	0 → K+2

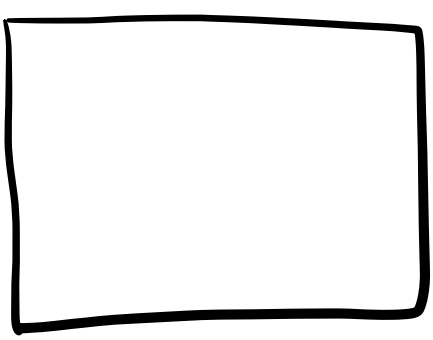
	0	1	2	3	4	5	6	7	8	
ar →	13	22	31	45	11	20	48	60	50	K = 4
sort →	11	13	20	22	31	45	48	50	60	

0-4 in heap

0-5 insert 20

0-6 insert 48

insert 60



idx : [0, K)

1. Insert first K+1 de into minHeap
2. ans [N]
3. ans.add(mh.ExtractMin())
4. for(i = K+1 ; i < n ; i++){
   mh.add(arr[i])
   ans.add(mh.ExtractMin())
   }
5. while(mH.size() > 0){
   ans.add(mh.ExtractMin())
   }

return ans

idx	
0	0 → K
1	0 → K+1
2	0 → K+2

TC: $O(K + (N-K)\log K + K\log K)$

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

TC: $O(N \log K)$ $\quad K < N$

SC: $O(K)$

Median - Middle value in sorted data

6 3 8 11 20 → Sort → 3 6 [8] 11 20

Median = 8

1 2 4 3 → Sort → 1 [2 3] 4

Median = $\frac{2+3}{2} = 2.5$

5. Find the median of stream

6 3 8 11 20

Median 6 4.5 6 7 8

9/2

Idea 1: For every new de, insert in ar[], sort it → median

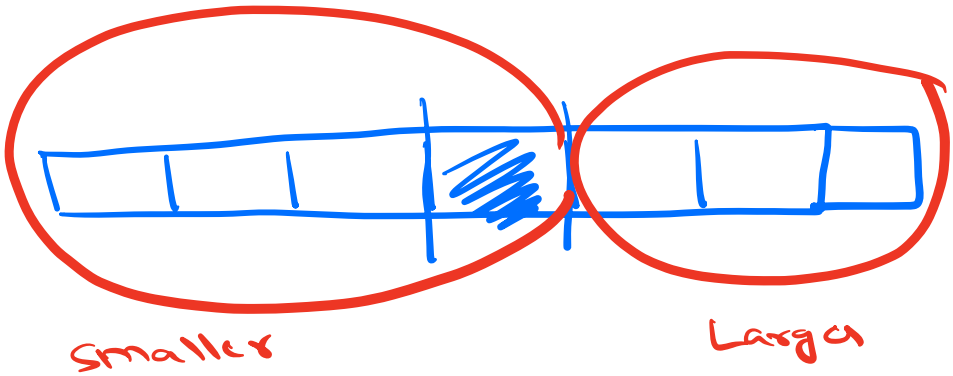
TC: $O(N \times N \log N)$

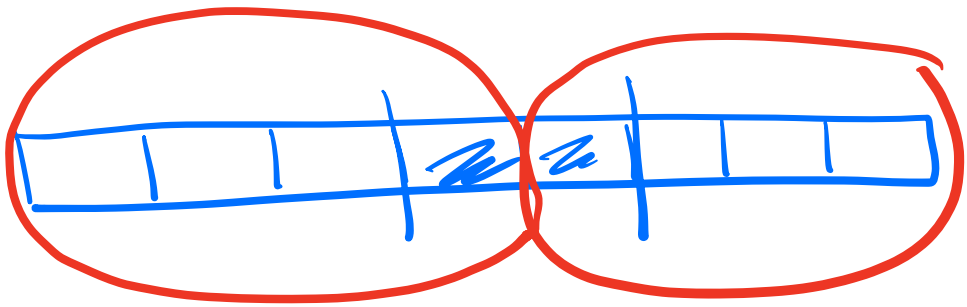
$O(N^2 \log N)$

Idea 2: For every new de, instead of sorting from scratch, simply insert new de at the correct pos in sorted array using insertion step.

TC: $O(N^2)$

Idea 3





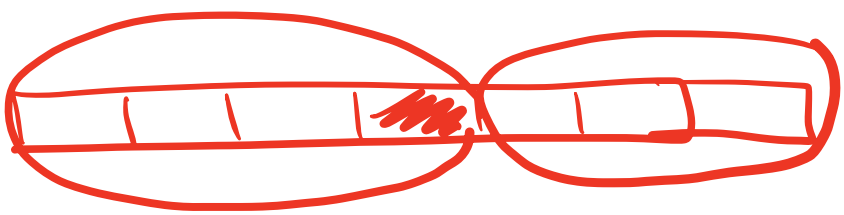
Max

Min

Max Heap

Min Heap

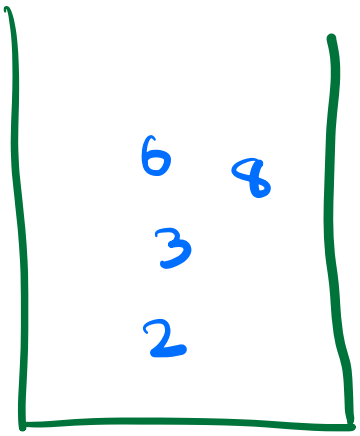
S	L	
8	7	15
8	8	16



Max

② size (S) − size (L) ≤ 1

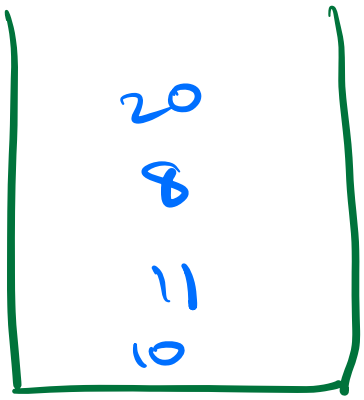
i	0	1	2	3	4	5	6	7	8	9
a:	6	3	8	11	20	2	10	8	13	50
	6	4.5	6	7	8	7	8	8		



S

Max Heap

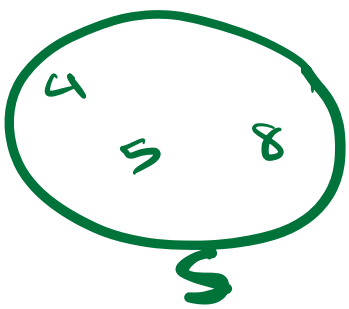
H_1



L

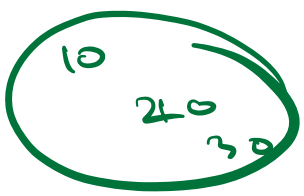
Min Heap

H_2



S

<



L

MaxHeap h1 (small)

MinHeap h2 (large)

h1.add(ar[0]) print(ar[0])

```
for (i = 1 ; i < n ; i++) {
    if (ar[i] > h1.getMax())
        h2.insert(ar[i])
    else
        h1.insert(ar[i])

    diff = h1.size() - h2.size()
    if (diff == 2) {
        // h1 has extra element
        h2.add(h1.extractMax())
    }
    else if (diff == -1) {
        h1.add(h2.extractMin())
    }
    // h has more el than L
    if (h1.size() > h2.size())
        print(h1.getMax())
    else {
        print((h1.getMax() + h2.getMin()) / 2)
    }
}
```

TC: O(N log N)

SC: O(N) → Max and Min Heap