Hashmap & Heap

DSA classes

Order of Content

1. → Hashmap & Heap → level 1 (without construction)
2. → Graphs - Lecture (1)+(2)
3. → Hashmap & Heaps + Arrays & Strings → level 2 + construction
   - # Chaining
   - # Open Addressing
   - # Good Hashfunction, Bad Hashfunction
4. → Bit Manipulation → Competitive
5. → Number Theory → Competitive
6. → Trees - level 2 of remaining lectures
   - (Revising DP)
   - DP on trees
   - Self Balancing BST (AVL)

Data Structures

	Arrays/Strings	# hh(PHR)
① add (insert) → O(1)	O(1) last, O(n) first	O(1) f, O(1) last
② remove (delete) → O(1)	O(1) last, O(n) first	O(1) f, O(1) last
③ update → O(1)	O(1) {indexing}	O(n) {indexing}
④ search (find) → O(1)	O(n) LS, O($\log_2 n$) BS	O(n) LS
⑤ display/Traverse	O(n)	O(n)
⑥ length/size	O(1)	O(1)
⑦ read (get) → O(1)	O(1) {indexing}	O(n) worst case


Team (String)
IPL Trophies count (Integer)
Team (String)
Years of winning (ArrayList<Integer>)
Java
Hashmap
key-value pairs
Read
Create
Update
Delete
Find
Class & objects
C++
Unordered_map
Python
Dictionary
Json
key: value
(String) Key → Country
(Long) Value = Population

Hashmap

HashMap<Key, Value> hm = new HashMap<>();

1. insert → put → O(1) (key not found)
2. update → put → O(1) (key found)
3. delete → remove → O(1)
4. find/read → get → O(1)
5. search → containsKey → O(1) (value/key)
6. size() → O(1) {k-v pairs}
7. display → Syso(hm); → O(n)
8. keyset → O(n) → Set<Key> → traverse

function, variable name

↓ ↓

Java: getOrDefault() populationOfCountry

C++: get_or_default() population_of_country

HashMap →
- Order of insertion ≠ Order of key-value pair
- ≠ Sorted order based on key/value

Unordered
no indexing

Number of Key-value pairs
= Number of keys

Unique Keys
↳ put → Insert (key ∄)
→ Update (key ∃)

1 key → 1 entry { No duplicates keys will be present }

Common Elements / Intersection

{ 2, 1, 1, 3, 4, 2, 5, 1, 2, 5 }

{ 1, 1, 4, 4, 4, 5, 5, 3 }

LC 349

{ 1, 3, 4, 5 }

LC 350

{ 1, 1, 3, 4, 5, 5 }

Brute force

Sort Both Arrays + Two pointer

$N_1 \log N_1 + N_2 \log N_2 + (N_1 + N_2)$

Optimized Hashmap

{ 2, 1, 1, 3, 4, 2, 5, 1, 2, 5 }

{ 1, 1, 4, 4, 4, 5, 5, 3 }

"1,4,5,3"

HashMap<Integer, Integer>

Element → Frequency

T, W, T → 9:45 PM - 11:45 PM (2hr)

S, S → 9 AM - 12 PM (3hr)

↓ 2-3 weeks

T, W, T → 9:15 PM - 11:15 PM (2hr)

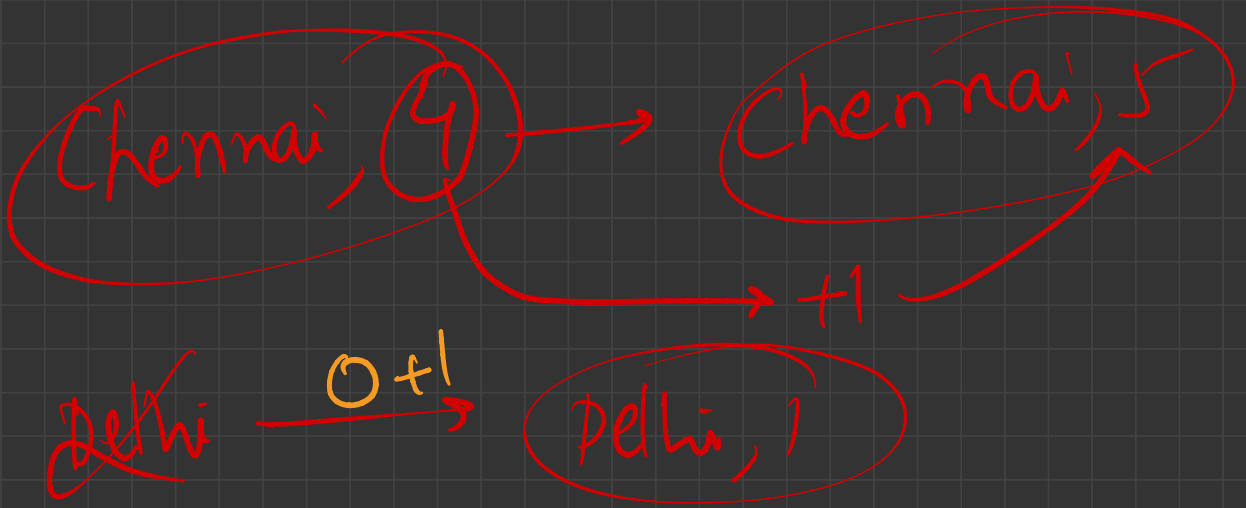
S, S → 9 AM - 12 PM (3hr)

```
if(freq.containsKey(nums1[i]) == true){
    freq.put(nums1[i], freq.get(nums1[i] + 1);
} else {
    freq.put(nums1[i], 1);
}
```

1

2

```
freq.put(nums1[i], freq.getOrDefault(nums1[i], 0) + 1);
```



349 LC

```
HashMap<Integer, Integer> freq = new HashMap<>();

for(int i=0; i<nums1.length; i++){
    // if(freq.containsKey(nums1[i]) == true){
    //     freq.put(nums1[i], freq.get(nums1[i] + 1);
    // } else {
    //     freq.put(nums1[i], 1);
    // }

    freq.put(nums1[i], freq.getOrDefault(nums1[i], 0) + 1);
}

ArrayList<Integer> intersection = new ArrayList<>();

for(int j=0; j<nums2.length; j++){
    if(freq.containsKey(nums2[j]) == true){
        intersection.add(nums2[j]);
        freq.remove(nums2[j]);
    }
}

int[] res = new int[intersection.size()];
for(int i=0; i<res.length; i++)
    res[i] = intersection.get(i);
return res;
```

N1

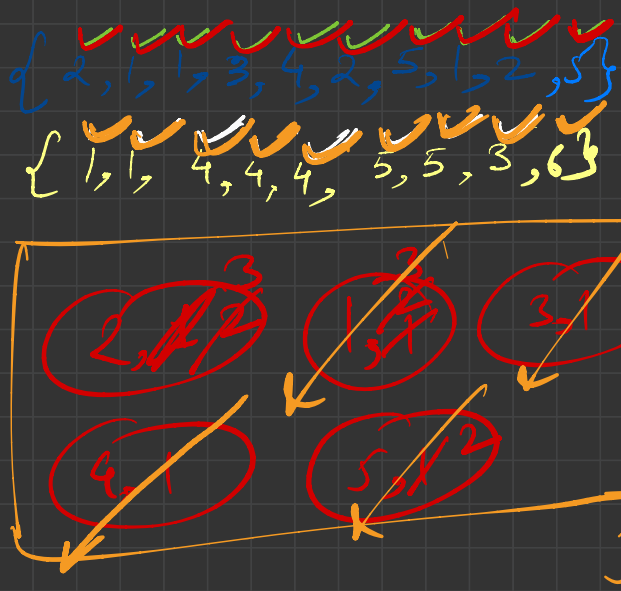
O(1)

O(1)

N2

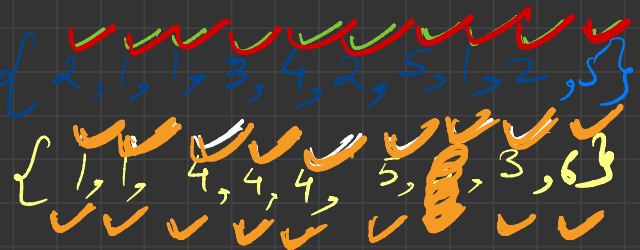
O(1)

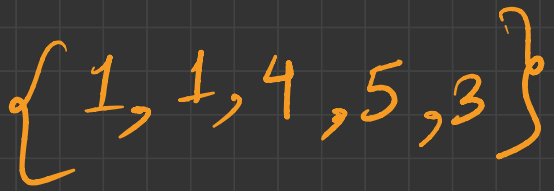
O(1)

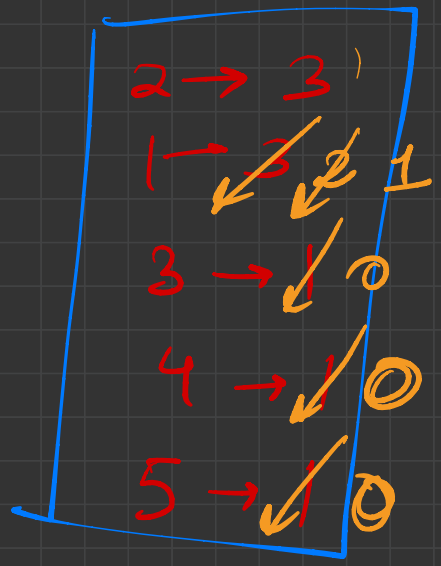


{1, 4, 5, 3}

O(N1+N2) Time , O(N1) extra Space (Hashmap)







freq

abcde

a→1, b→1, c→1, d→1

e→1

{a→{a,b,c,d,e}}

aaaaa

a→s

s→{a}

```java
HashMap<Integer, Integer> freq = new HashMap<>();

for(int i=0; i<nums1.length; i++)
    freq.put(nums1[i], freq.getOrDefault(nums1[i], 0) + 1);

ArrayList<Integer> intersection = new ArrayList<>();

for(int j=0; j<nums2.length; j++){
    if(freq.containsKey(nums2[j]) == true && freq.get(nums2[j]) > 0) {
        intersection.add(nums2[j]);
        freq.put(nums2[j], freq.getOrDefault(nums2[j], 0) - 1);
    }
}

int[] res = new int[intersection.size()];
for(int i=0; i<res.length; i++)
    res[i] = intersection.get(i);
return res;
```

Sort String by Frequency

Frequency Sort

a b c a b d d c d b e

0 1 2 3 4 5 6 7 8 9 10

N = 11

HashMap<Character, Integer> freq = new HashMap<>();

key, value

get-value

a → 1 2

b → 1 2 3

c → 1 2

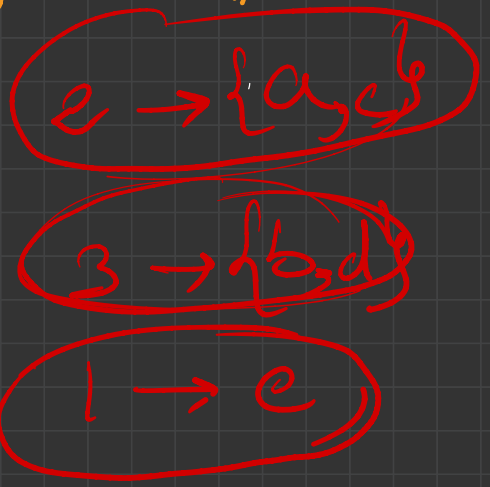
d → 1 2 3

e → 1

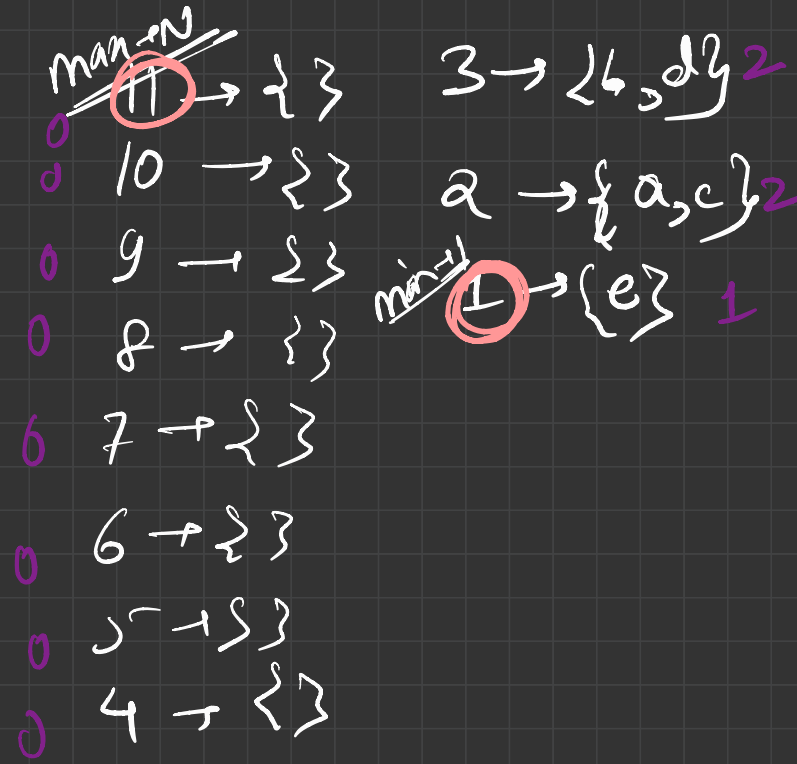
Expected Output: bbb ddd aa cc e

3 2 1

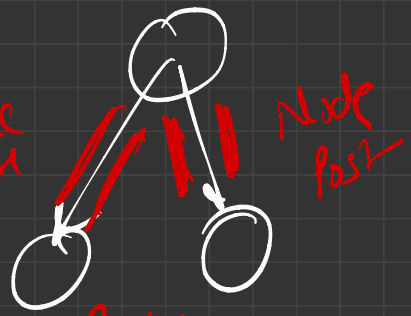
Hashmap <Integer, Arraylist <Character>>



bbb ddd aac e



Recursion
↳ DFS
↳ Trees



Graph
① Hashmap (Use eg)
② Trees → DFS { Inorder & Post }
③ Arraylist / OOPS

```java
HashMap<Character, Integer> freq = new HashMap<>();
for(int i=0; i<s.length(); i++){
    char ch = s.charAt(i);
    freq.put(ch, freq.getOrDefault(ch, 0) + 1);
}

HashMap<Integer, ArrayList<Character>> rev = new HashMap<>();
for(Character ch: freq.keySet()){
    int f = freq.get(ch);
    if(rev.containsKey(f) == false)
        rev.put(f, new ArrayList<>());
    rev.get(f).add(ch);
}

StringBuilder res = new StringBuilder("");
for(int f=s.length(); f>=1; f--){
    if(rev.containsKey(f) == true){
        for(Character ch: rev.get(f)){
            for(int i=0; i<f; i++)
                res.append(ch);
        }
    }
}

return res.toString();
```

→ O(N)

→ O(26+26)

→ O(N)

Time → O(N)
Space → O(N+26)

C → I

a → 1/2 b → 2/3 c → 1/2 d → 1/3 e → 1

Int → ALC7

2 → a, c 3 → b, d 1 → e

res = "bbdddaaccce"

(1) → 3, 2, 1

11 → {} 0 8 → {} 0 5 → {} 0 2 → {a, c} 2

10 → {} 0 7 → {} 0 4 → {} 0

9 → {} 0 6 → {} 0 3 → {b, d} 2 1 → {e} 1

N + 26 + 26 + N = O(N)

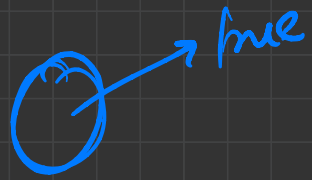
11 + 5 + (3+3+2+2+1)

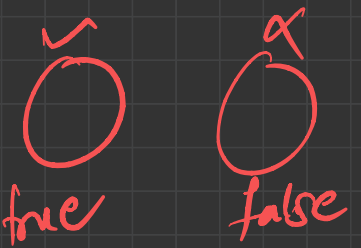
b → 3 a → 2 e → 1 ⇒ 11 total

d → 3 c → 2

HashMap < Key , Boolean > hm ⟹ HashSet<Key>

value

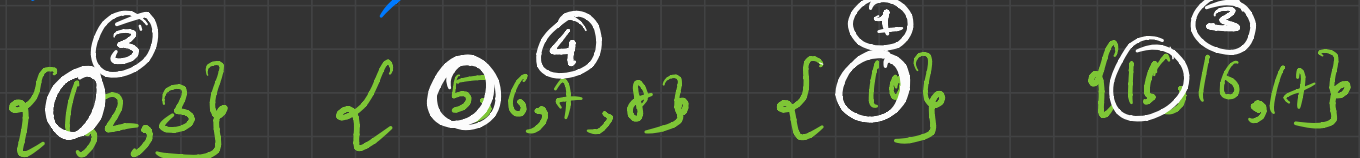




Longest Consecutive Sequence

RC-128

{ 10, 2, 8, 15, 1, 5, 7, 6, 17, 16, 3 }



① Sorting → Time → NlogN , Space → O(1)

{1, 2, 3, 5, 6, 7, 8, 10, 15, 16, 17}

\+

TwoPointer → N

O(NlogN + N) = O(NlogN)

② Hashmap

↓

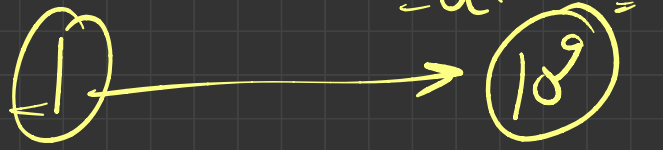
Each elements if it starts the sequence, Longest sequence?

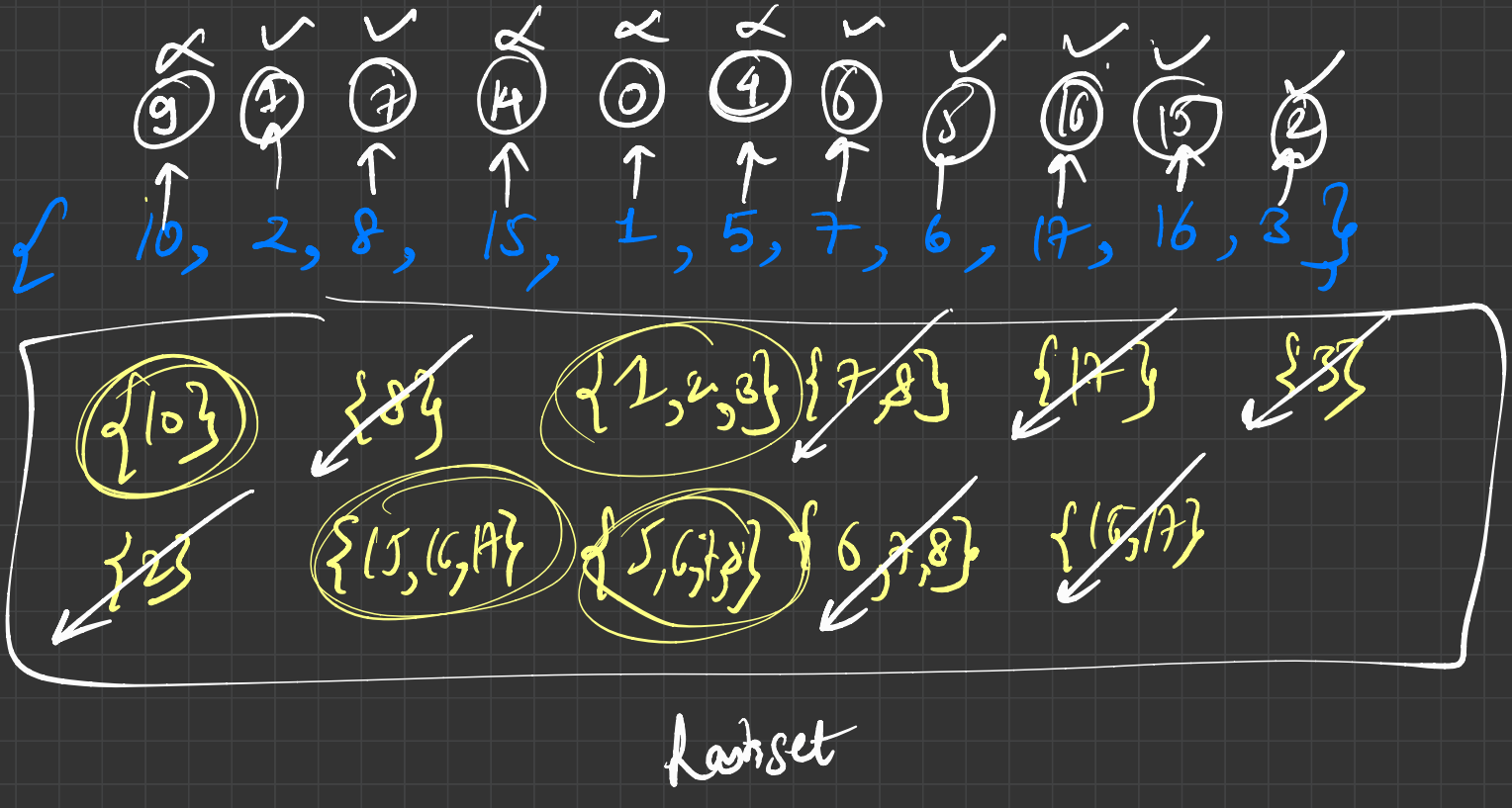
③ Min → Max

No No

O(Max_n − Min_n)

= O(10^9 + 10^9)



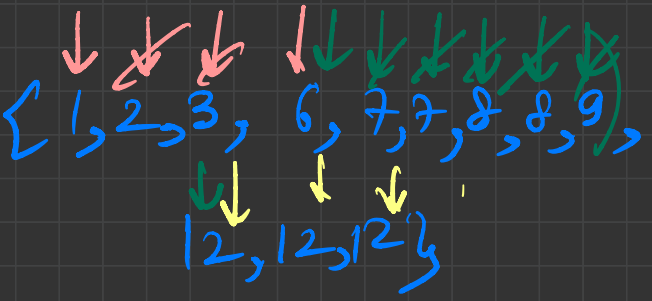

9 1 7 14 0 4 6 5 10 15 2
{ 10, 2, 8, 15, 1, 5, 7, 6, 17, 16, 3 }
{10}
{8}
{1,2,3}
{7,8}
{17}
{3}
{2}
{5,6,14}
{5,6,7,8}
{6,7,8}
{16,17}
Rankset

Traversals

1. Keys → HashSet
2. Delete keys which cannot start a new sequence
3. Disjoint sequence → longest

```java
public int longestConsecutive(int[] nums) {
    if(nums.length == 0) return 0;

    Arrays.sort(nums);

    int left = 0, max = 1;
    while(left < nums.length){

        int right = left + 1, count = 1;
        while(right < nums.length && nums[right] <= nums[right - 1] + 1){
            if(nums[right] == nums[right - 1] + 1)
                count++;
            right++;
        }

        max = Math.max(max, count);
        left = right;
    }

    return max;
}
```

① Sorting + Two pointer

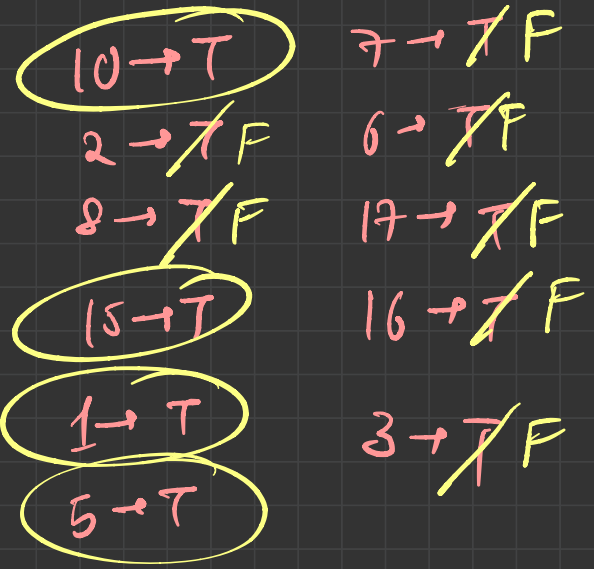


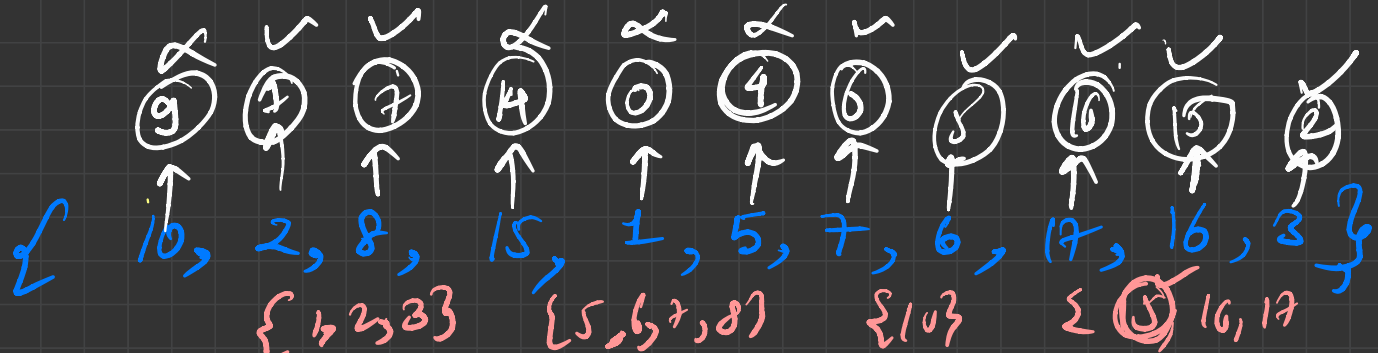
count = ~~X~~ ~~2~~ ~~3~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ 1

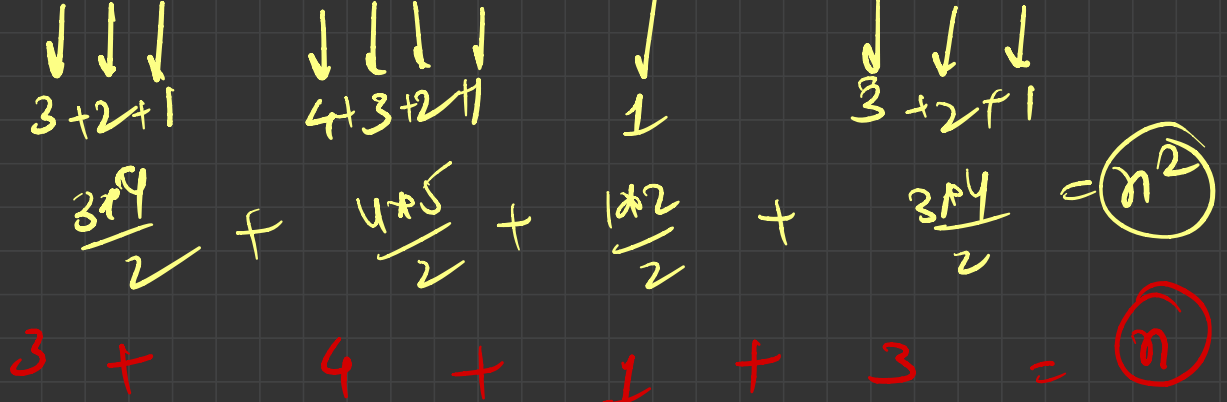
max = ~~X~~ 3 4

Time → NlogN, Space → O(1)

```java
// Unique Key
HashMap<Integer, Boolean> isStarting = new HashMap<>();

// 1. Take All Elements as Potential Starting Sequence
for(int i=0; i<nums.length; i++){
    isStarting.put(nums[i], true);
}

// 2. Remove Elements which cannot start a new Sequence
for(int i=0; i<nums.length; i++){
    if(isStarting.containsKey(nums[i] - 1) == true){
        // We Will Join the Previous Sequence,
        // Hence, we cannot start the sequence
        isStarting.put(nums[i], false);
    }
}

int max = 0;
```








```
// 3. Finding the Maximum Length among all Disjoint Sets
for(int start: isStarting.keySet()){

    if(isStarting.get(start) == true){
        int curr = 1;
        for(int end=start+1; isStarting.containsKey(end) == true; end++){
            curr++;
        }

        max = Math.max(max, curr);
    }

}

return max;
```

Binary Heap vs Priority queue

Binary Heap → concrete data structure

Priority queue → abstract data type

- (LIFO) Stack
- (FIFO) Queue (min-max)
- BST
- Binary Heap

- ↳ highest-priority get O(1)
- ↳ remove(HP) → O(log N)
- ↳ insert(HP) → O(log N)

Order → (LIFO)

Highest Priority → last (Insertion order)

Least Priority → First

get last → O(1)
remove last → O(1)
insert last → O(1)

} Stack

Order → (FIFO)

Least Priority → last (Insertion order)

Highest Priority → First

get First
remove First
add Last

} Queue

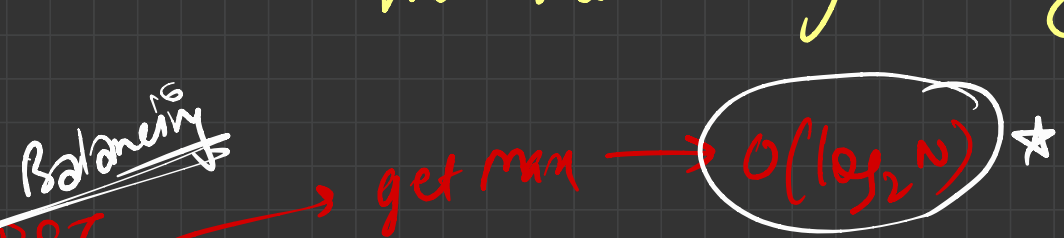
10, 20, 30, 40, 50

min value → Least Priority

values

max value → Highest Priority

Sorted Array



O(1)

,

O(1)

,

O(N)

Binary Heap

getHP() → O(1)

removeHP() → $O(\log_2 n)$

Insert () → $O(\log_2 n)$

Priority Queue (Binary Heap)

- → upheapify, downheapify,
- → efficient construction, heapsort

- → Generic
- → objects ↔ Comparable / Comparator

⇒ PriorityQueue<Datatype> pq = new PriorityQueue<>();

① Insert

② Remove → Highest Priority

③ Get → Highest Priority

④ Update
↳ remove() + insert()

Insert (add) → $O(\log_2 N)$

Remove → $O(\log_2 N)$

Get (peek) → $O(1)$

N = number of elements present in heap

length → $O(1)$
↳ size()

Min Heap

Comparison

Max Heap

Min element → highest priority (ascending)

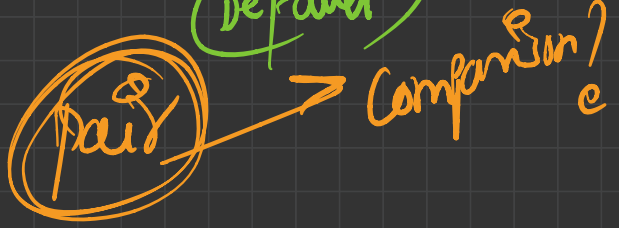
Max element → highest priority

value → Wrapper class ← Integer, Long, Float, Double

String (lexicographical)

Java → Min Heap (Default)

C++ → Max Heap (Default)



pair<int, int>

↳ default ✓

If N elements are inserted (1 at a time)

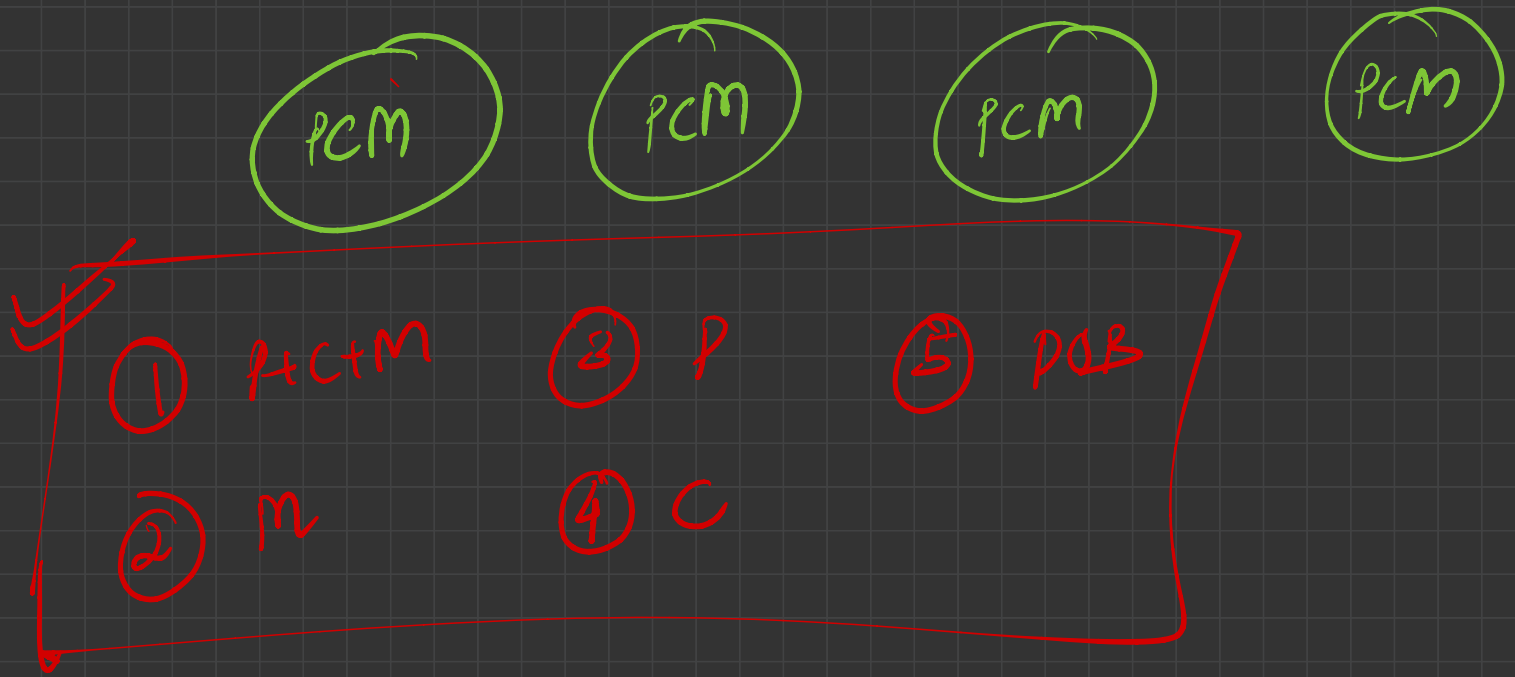
$\Rightarrow \log 1 + \log 2 + \log 3 + \cdots \log N$

$= \log(1 * 2 * 3 * 4 \cdots N) = \log N!$

$N! \approx N^N$

$= \log N^N$

$= O(N \log N)$



K largest Elements

[50, 20, 100, 30, 90, 10, 60, 70, 80, 40]

K = 4

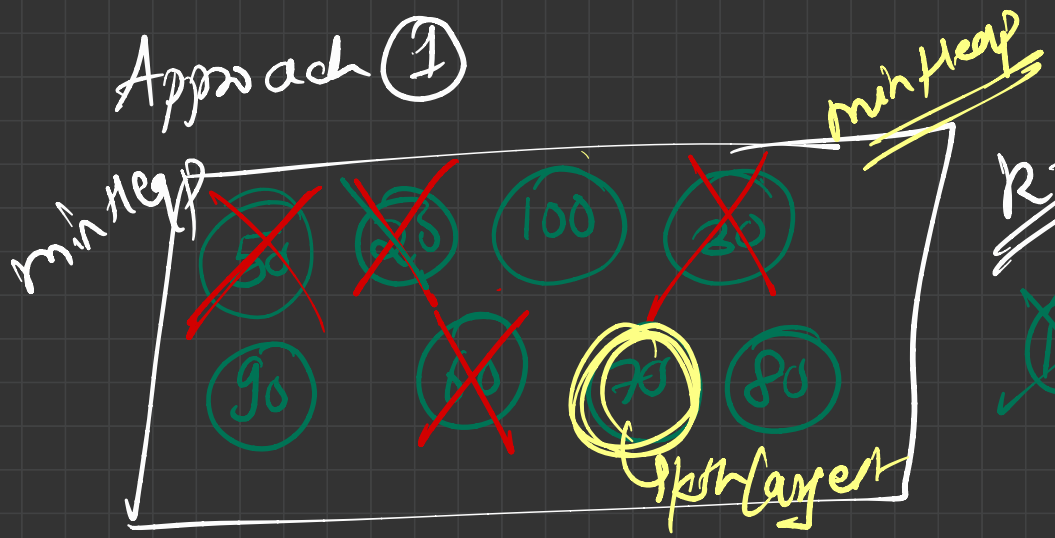
Brute force → Sorting ⭐

Time → O(nlogn)

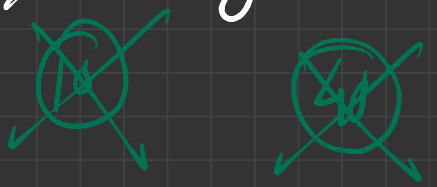
space → O(1)

Priority Queue

Approach ①



K=4 largest Elements



→ First K elements in PQ (minHeap)

⟷ Each (N-K) elements → If curr > min
→ insert(curr)
→ remove()

else do nothing

```java
//Function to return k largest elements from an array.
public static ArrayList<Integer> kLargest(int arr[], int n, int k)
{
    PriorityQueue<Integer> q = new PriorityQueue<>();
    for(int i=0; i<n; i++){
        if(i < k){
            q.add(arr[i]);
        } else if(q.peek() < arr[i]) {
            q.remove();
            q.add(arr[i]);
        }
    }

    ArrayList<Integer> res = new ArrayList<>();
    while(q.size() > 0){
        res.add(q.remove());
    }
    Collections.reverse(res);
    return res;
}
```

nlogk

+

klogk

k

Time

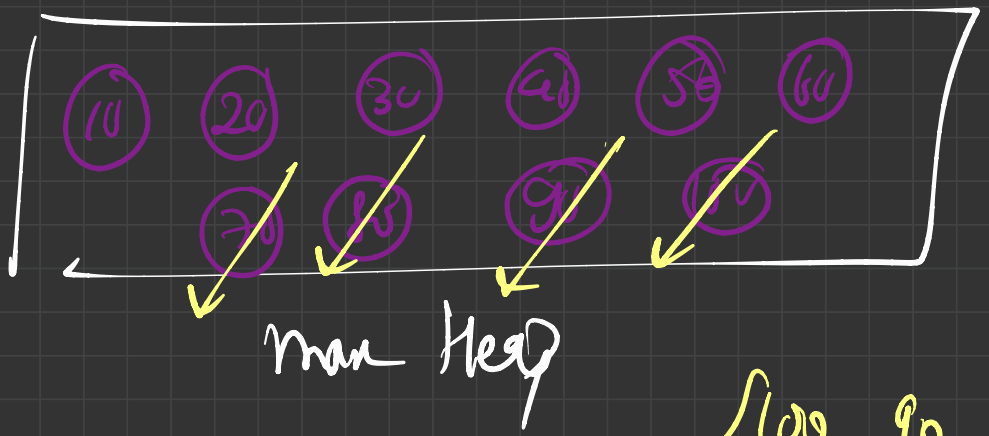
$(N+k)\log k$

$\approx n \log k$

Space

$O(k)$

Approach (2)



If 1 at a time $\rightarrow O(N \log N)$

{ $O(N)$ efficiently Insert

+ { Array is inserted once }

kth largest

{100, 90, 80, 70}

k removals $\rightarrow k \log N$

Total time $= O(N + k \log N)$

```java
// Time - O(N + KlogN) MAX HEAP APPROACH
PriorityQueue<Integer> q = new PriorityQueue<>(Collections.reverseOrder());
for(int i=0; i<n; i++){
    q.add(arr[i]);
}

ArrayList<Integer> res = new ArrayList<>();
while(res.size() < k){
    res.add(q.remove());
}
return res;
```

Time → O(N + KlogN)

Space → O(N)

lc215 Kth largest Elements

App 1 (min Heap)

Time → O(n log K + K log N)

insert 1 at a time

& Remove k elements

Space → O(n)

log n

```
public int findKthLargest(int[] arr, int k) {
    PriorityQueue<Integer> q = new PriorityQueue<>(Collections.reverseOrder());
    for(int i=0; i<arr.length; i++){
        q.add(arr[i]);
    }

    int kthLargest = q.peek();
    while(k-- > 0){
        kthLargest = q.remove();
    }
    return kthLargest;
}
```

App 2

max Heap

(addAll) insert removal

Time → O(N + K log N)

Space → O(N)

Ans

```
public int findKthLargest(int[] arr, int k) {
    ArrayList<Integer> temp = new ArrayList<>();
    for(int val: arr) temp.add(val);

    PriorityQueue<Integer> q = new PriorityQueue<>(Collections.reverseOrder());
    q.addAll(temp);

    int kthLargest = q.peek();
    while(k-- > 0){
        kthLargest = q.remove();
    }
    return kthLargest;
}
```

App ③ Min Heap

Time → NlogK , Space → O(K)

7ms

```
public int findKthLargest(int[] arr, int k) {
    PriorityQueue<Integer> q = new PriorityQueue<>();
    for(int i=0; i<arr.length; i++){
        if(i < k){
            q.add(arr[i]);
        } else if(q.peek() < arr[i]) {
            q.remove();
            q.add(arr[i]);
        }
    }

    return q.peek();
}
```

App 4 QuickSelect { similar to quick sort }

O(N) partitioning

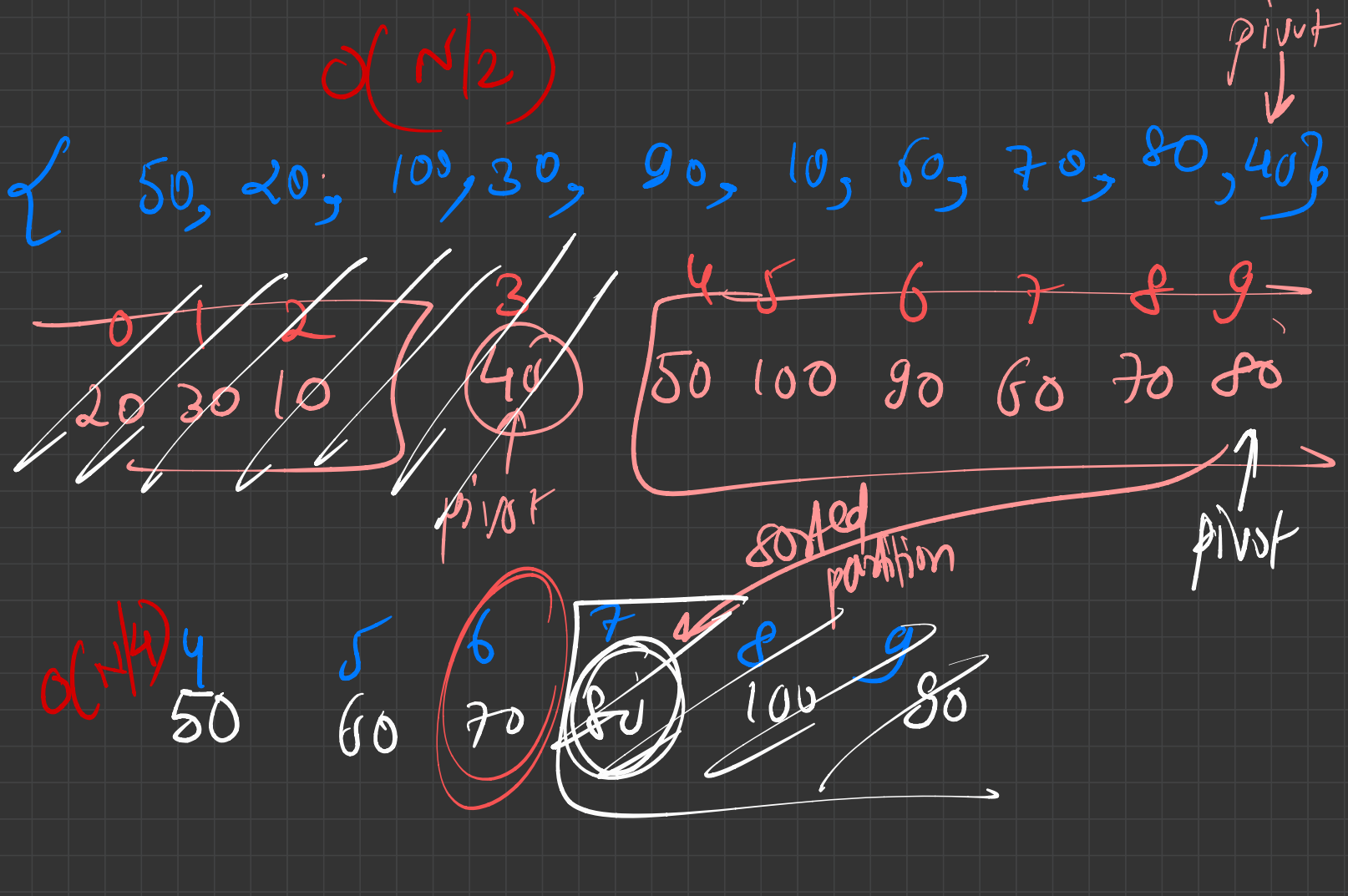
{ 50, 20, 100, 30, 90, 10, 60, 70, 80, 40 }

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100

↑ smaller

↑ Largest

N = 10, k = 4, kth Largest = (N − k)


O(N/2)
pivot
{ 50, 20, 100, 30, 90, 10, 60, 70, 80, 40 }
0 1 2 3
20 30 10 40
pivot
4 5 6 7 8 9
50 100 90 60 70 80
pivot
sorted partition
O(N/4)
4 5 6 7 8 9
50 60 70 90 100 80

$$\text{Total time} = N + \frac{N}{2} + \frac{N}{4} + \cdots \text{O}$$

$$= N\left\{1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \cdots\right\}$$

$$= N\{2\} \Rightarrow \boxed{O(N)}$$

↑ average case Quickselect

Worst case

$(n-1) + (n-2)$

$N + (N-1) + \cdots 1$

$+ \cdots = O(N^2)$

```java
public int partition(int[] arr, int l, int r){
    int j = l;
    for(int i=l; i<=r; i++){
        if(arr[i] <= arr[r]){
            int temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
            j++;
        }
    }
    return j - 1;
}

public int quickSelect(int[] arr, int l, int r, int k){
    if(l == r) return arr[l];

    int pivot = partition(arr, l, r);
    if(pivot == arr.length - k) return arr[pivot];
    if(pivot > arr.length - k) return quickSelect(arr, l, pivot - 1, k);
    return quickSelect(arr, pivot + 1, r, k);
}

public int findKthLargest(int[] arr, int k) {
    return quickSelect(arr, 0, arr.length - 1, k);
}
```

K closest Points to Origin

$(x, y) \rightarrow \sqrt{x^2+y^2}$

$(6,8) = 10$ $\quad \sqrt{6^2+8^2}$

$(3,2) = \sqrt{13} \approx 3.6$ $\quad \sqrt{3^2+2^2}$

$(-4,1) = \sqrt{17} \approx 4.1$ $\quad \sqrt{(-4)^2+1^2}$

$(1,-1) = \sqrt{2}$ $\quad \sqrt{1^2+(-1)^2}$

$(1,4) = \sqrt{17} \approx 4.1$ $\quad \sqrt{1^2+4^2}$

$(8,6) = 10$ $\quad \sqrt{8^2+6^2} = \sqrt{100}$

$(12,5)$ $\quad \sqrt{12^2+5^2} = 13$

$(0,0)$ $\quad \sqrt{0^2+0^2} = 0$

$(3,4)$ $\quad \sqrt{3^2+4^2} = 5$

$(-5,12)$ $\quad \sqrt{(-5)^2+12^2} = 13$

$K = 4$

$(0,0)$ 0 dist

$(1,-1)$ $\sqrt{2}$ dist

$(-4,1)$ $\sqrt{17}$ dist

$(1,4)$ $\sqrt{17}$ dist

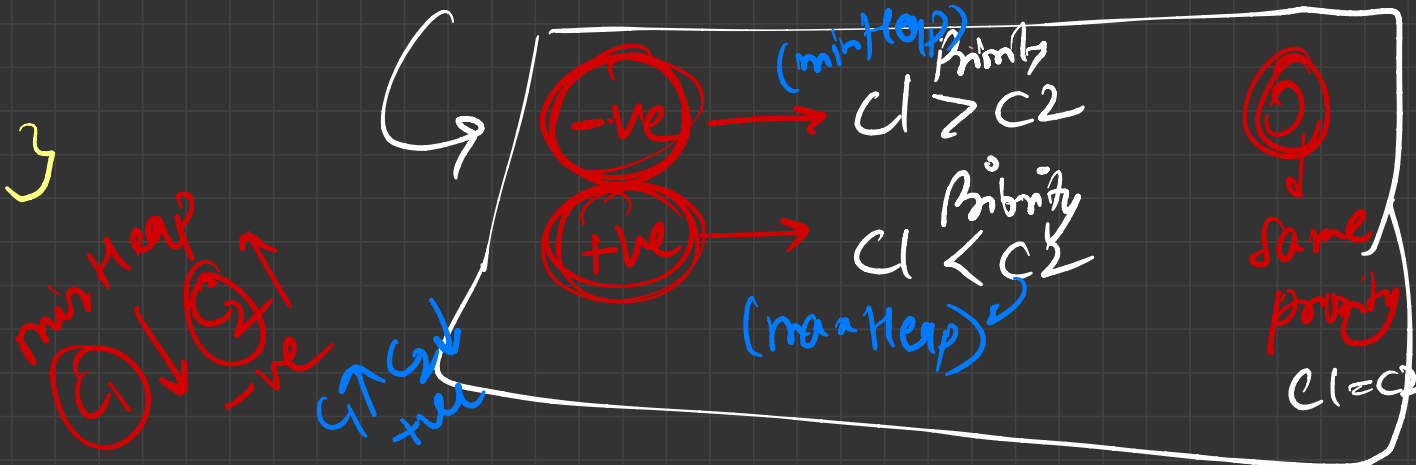
Euclidean distance $= \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

Manhatten Distance $= |x_2-x_1| + |y_2-y_1|$

Comparator

Function → 2 objects → Compare

```
class euclidean implements Comparator<int[]> {
    @Override
    public int compare(int[] c1, int[] c2){
    }
}
```



Closest Point to origin

↳ Euclidean Distance to origin

↳ $\sqrt{(x-0)^2 + (y-0)^2} = \sqrt{x^2+y^2}$

$c_1(\sqrt{x^2+y^2}) \overset{HP}{<} c_2(\sqrt{x^2+y^2})$

return -ve;

min Heap

$c_1(\sqrt{x^2+y^2}) > c_2(\sqrt{x^2+y^2})$ (HP)

return +ve;

$c_1(\sqrt{x^2+y^2}) == c_2(\sqrt{x^2+y^2})$

return 0;

```java
static class Pair{
    public int x, y;
}

static class Euclidean implements Comparator<Pair> {
    public int compare(Pair c1, Pair c2){
        double d1 = Math.sqrt(c1.x * c1.x + c1.y * c1.y);
        double d2 = Math.sqrt(c2.x * c2.x + c2.y * c2.y);

        if(d1 < d2){
            // Smaller Value -> Higher Priority
            return -1;
        } else if(d1 > d2){
            // Higher Value -> Lower Priority
            return 1;
        } else return 0;
    }
}
```

overrided

Interface

Euclidean

d1 < d2 → c1 is closer to c2 → c1 Higher Priority

d1 > d2 → c2 is closer than c1 → c2 Higher Priority

return 0 → c1 == c2 → same Priority

+ve → c1 Lower Priority, c2 High Priority

-ve → c1 Higher Priority, c2 lower Priority

0 → c1 = c2 Priority

```
public int[][] kClosest(int[][] points, int k) {
    PriorityQueue<Pair> q = new PriorityQueue<>(new Euclidean());

    for(int i=0; i<points.length; i++){
        Pair p = new Pair();
        p.x = points[i][0];
        p.y = points[i][1];
        q.add(p);
    }

    int[][] res = new int[k][2];
    int count = 0;
    while(count < k){
        Pair closest = q.remove();
        res[count][0] = closest.x;
        res[count][1] = closest.y;
        count++;
    }

    return res;
}
```

class → object → parameter → PQ constructor

Insertion → NlogN

Removal → KlogN

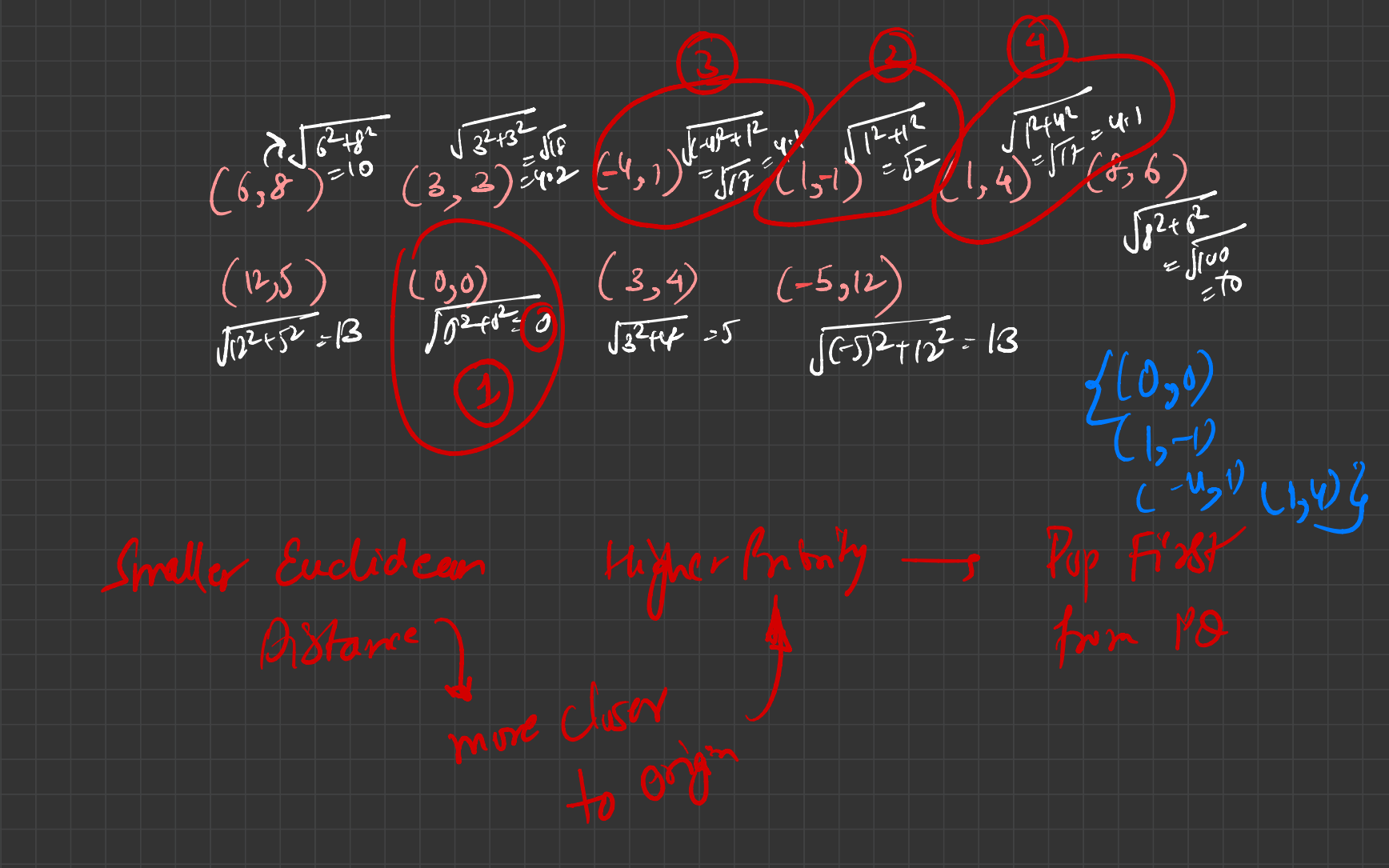
Time → O(NlogN) + O(KlogN)

Insertion

Removal

Auxiliary Space / Extra Space → O(N)

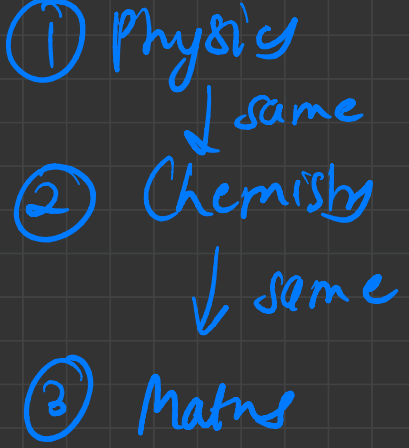
Priority Queue


(6,8) $\sqrt{6^2+8^2}=10$
(3,3) $\sqrt{3^2+3^2}=\sqrt{18}=4.2$
(-4,1) $\sqrt{(-4)^2+1^2}=\sqrt{17}=4.1$
(1,-1) $\sqrt{1^2+1^2}=\sqrt{2}$
(1,4) $\sqrt{1^2+4^2}=\sqrt{17}=4.1$
(8,6) $\sqrt{8^2+6^2}=\sqrt{100}=10$
(12,5) $\sqrt{12^2+5^2}=13$
(0,0) $\sqrt{0^2+0^2}=0$
(3,4) $\sqrt{3^2+4^2}=5$
(-5,12) $\sqrt{(-5)^2+12^2}=13$
3
2
4
1
{(0,0) (1,-1) (-4,1) (1,4)}
Smaller Euclidean Distance → more closer to origin
Higher Priority → Pop First from PQ

Marks of PCM

You are provided with marks of **N** students in Physics, Chemistry and Maths.
Perform the following 3 operations:

- Sort the students in Ascending order of their Physics marks.
- Once this is done, sort the students having same marks in Physics in the descending order of their Chemistry marks.
- Once this is also done, sort the students having same marks in Physics and Chemistry in the ascending order of their Maths marks.



Input:
N = 10
phy[] = {4 1 10 4 4 4 1 10 1 10}
chem[] = {5 2 9 6 3 10 2 9 14 10}
math[] = {12 3 6 5 2 10 16 32 10 4}

1 1 1 4 4 4 4 10 10
1 4 2 2 10 6 5 3 10 9
10 3 16 10 5 12 2 4 5


```
class MarksCmp implements Comparator<Marks>{

public int compare(Marks s1, Marks s2){
    if(s1.physics < s2.physics){
        return -1;
    }else if(s1.physics > s2.physics){
        return +1;
    }else{
```

HP

HP

HP

```
if( s1.chem < s2.chem){
    return +1;
} else if (s1.chem > s2.chem){
    return -1;
} else {
    if( s1.name( s2.name)
        return -1;
    else return +1;
}
}
```

```
class MarksCmp implements Comparator<Marks>{
    public int compare(Marks s1, Marks s2){
        if(s1.phy < s2.phy) return -1;
        else if(s1.phy > s2.phy) return +1;
        else {
            if(s1.chem < s2.chem) return +1;
            else if(s1.chem > s2.chem) return -1;
            else {
                if(s1.maths < s2.maths) return -1;
                return +1;
            }
        }
    }
}
```

MinHeap (Increasing)

MaxHeap (Decreasing)

```java
PriorityQueue<Marks> q = new PriorityQueue<>(new MarksCmp());

for(int i=0; i<N; i++){
    Marks stud = new Marks(phy[i], chem[i], math[i]);
    q.add(stud);
}

int idx = 0;
while(q.size() > 0){
    Marks stud = q.remove();
    phy[idx] = stud.phy;
    chem[idx] = stud.chem;
    math[idx] = stud.maths;
    idx++;
}
```

Insert
NlogN

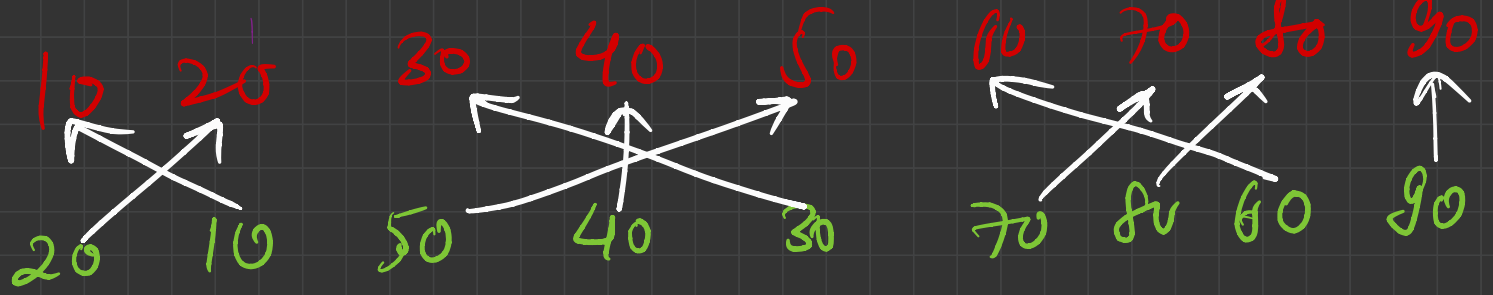
Removal
NlogN

O(NlogN)

Custom Sorting (Heap Sort)

Nearly Sorted Array → Sort

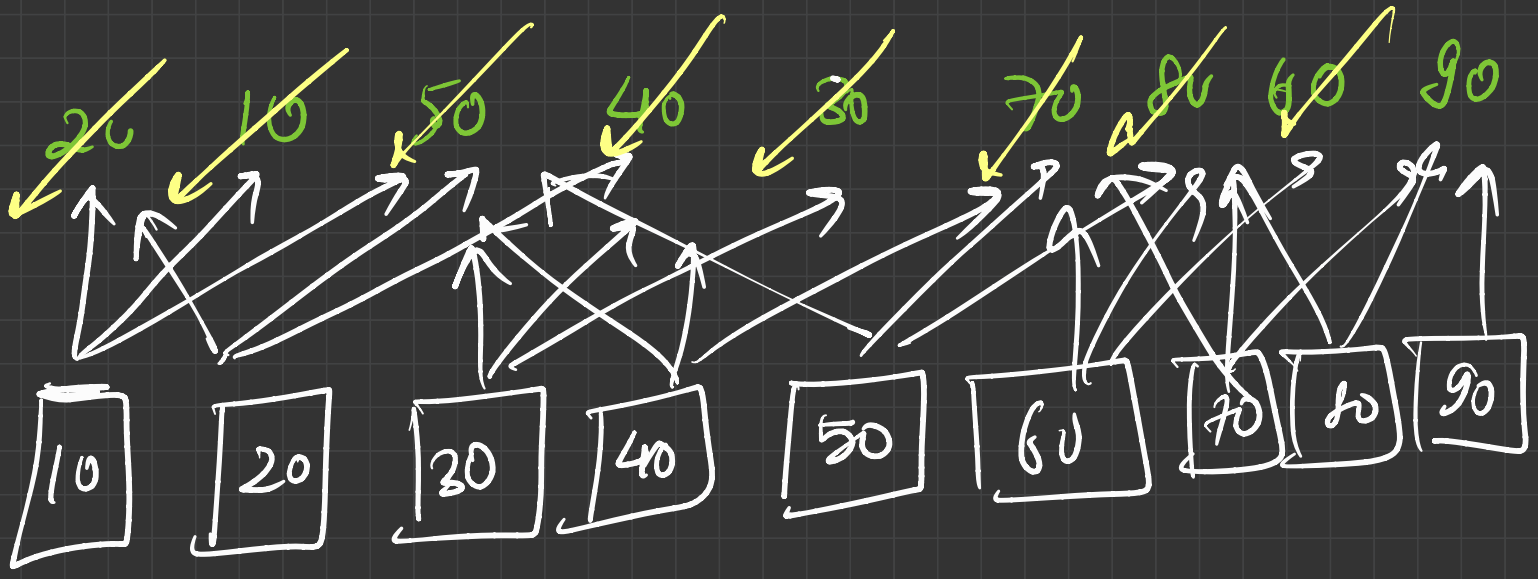
→ Atmost distance b/w actual position & sorted posⁿ

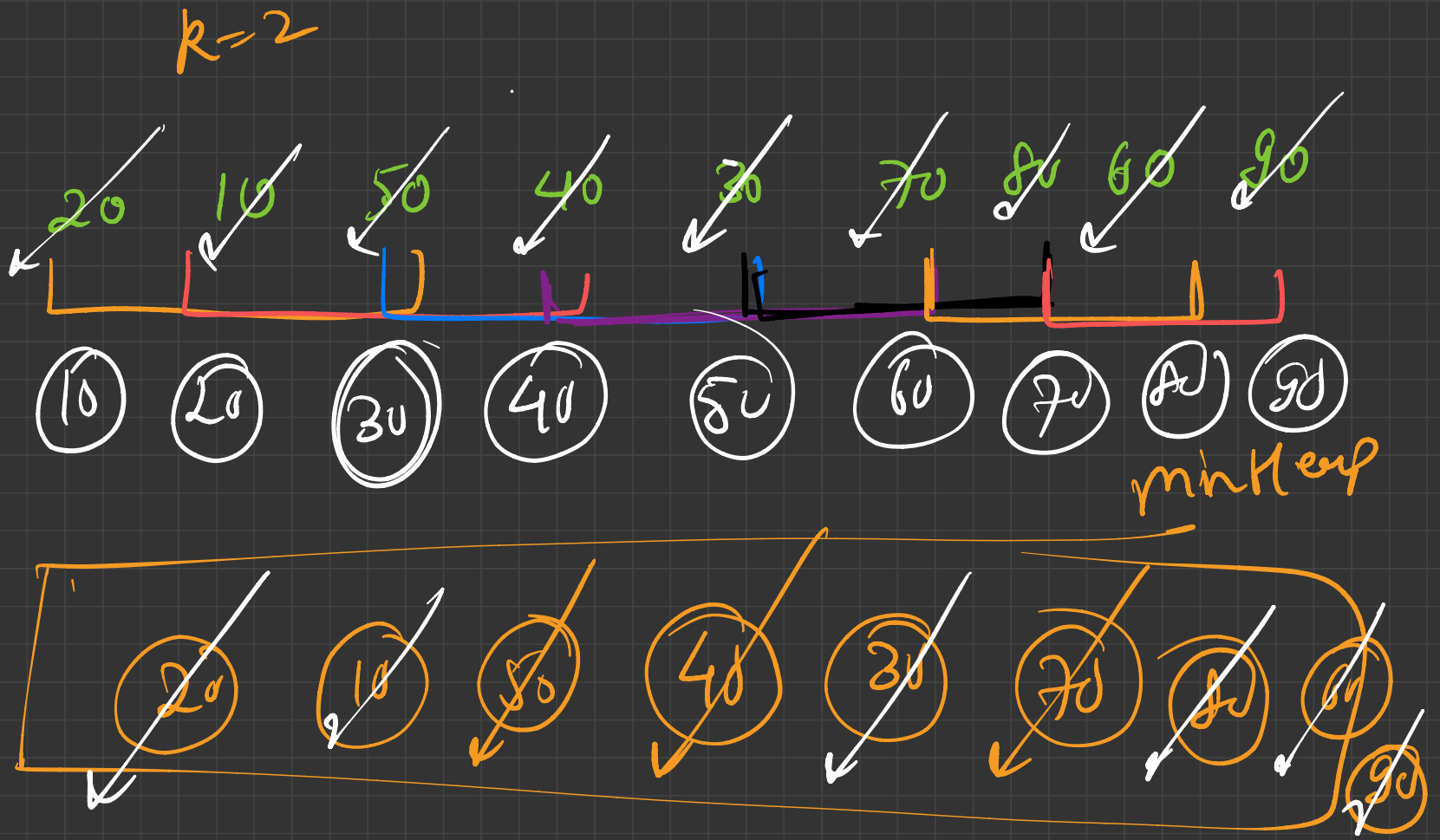


Bruteforce → Unsorted sorted → Sort

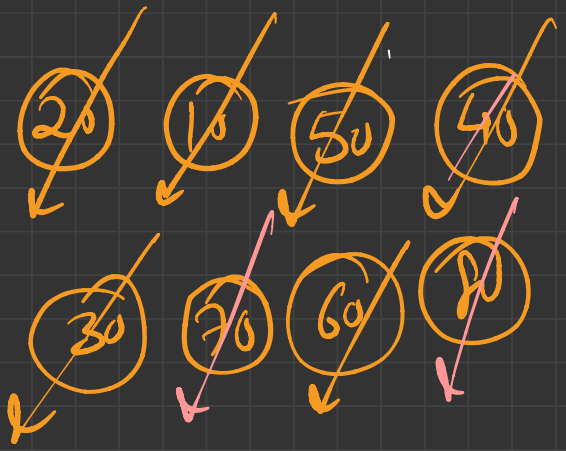
→ Time → O(nlogn)

R=2




K=2
20
10
50
40
30
70
80
60
90
10
20
30
40
50
60
70
80
90
minHeap
20
10
50
40
30
70
80
60
90

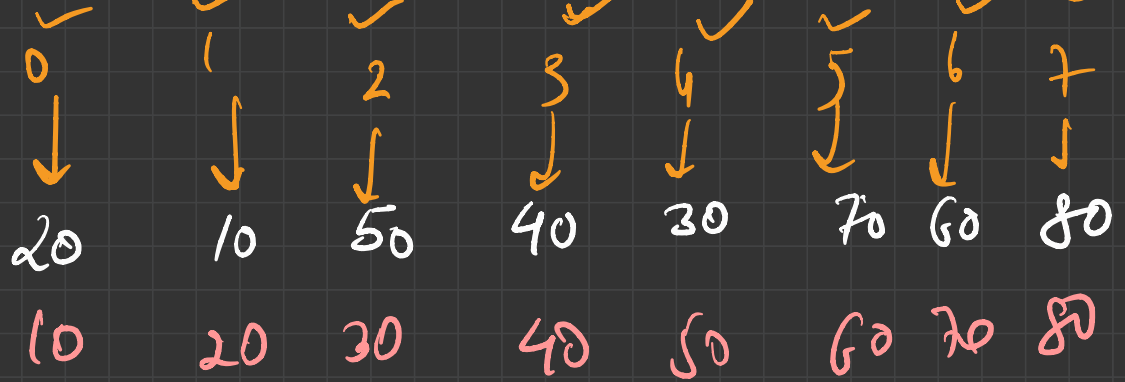
```java
PriorityQueue<Integer> q = new PriorityQueue<>();
for(int i=0; i<n; i++){
    q.add(arr[i]);
    if(i >= k){
        System.out.print(q.remove() + " ");
    }
}

while(q.size() > 0){
    System.out.print(q.remove() + " ");
}
```



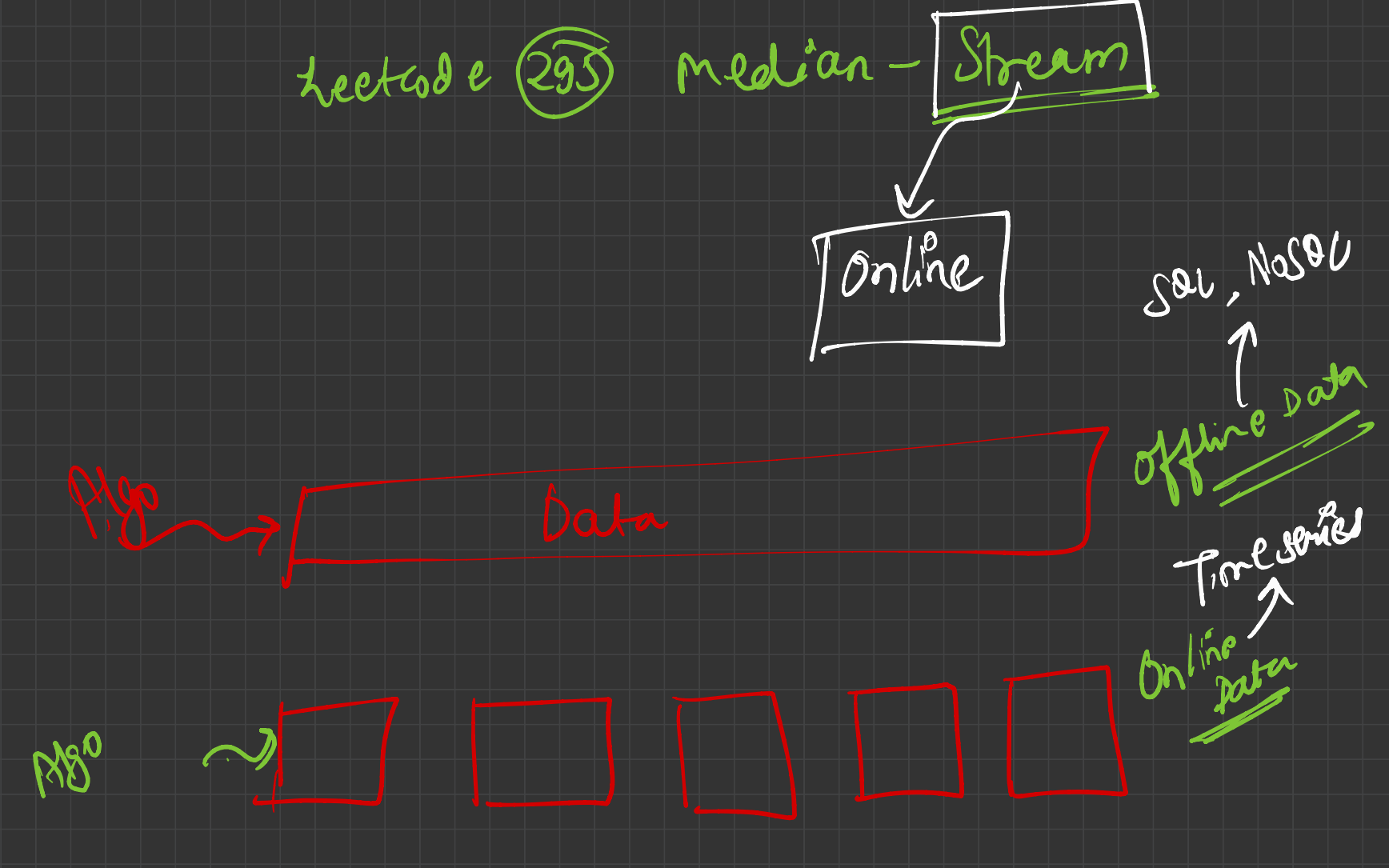
O(NlogK) + KlogK
Time

O(K) PQ
Space

K=2



Leetcode 295 Median - Stream



Median in Data Stream

Approach (1) ArrayList operation

[40] → 1 operation

median → O(1) → 40

[40, 70] → 1 operation

Median → O(1) → 55

[40, 50, 70] → 2 operation

median → O(1) → 50

Time Complexity

Insertion → worst-case O(N)

GetMedian → O(1)

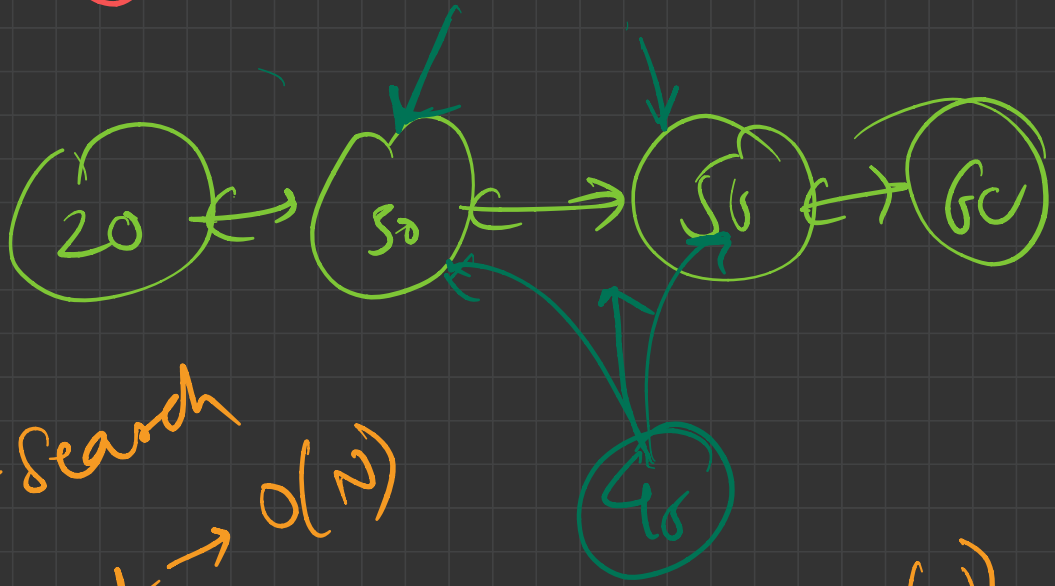
[40, 50, 70, 80] → 1 operation

median → O(1) → 60

[10 40 50 70 80] → 5 operation

median → O(1) → 50

Approach ② Linked List (Doubly)



linear search
L/r/sort → O(N)

get median → O(N)

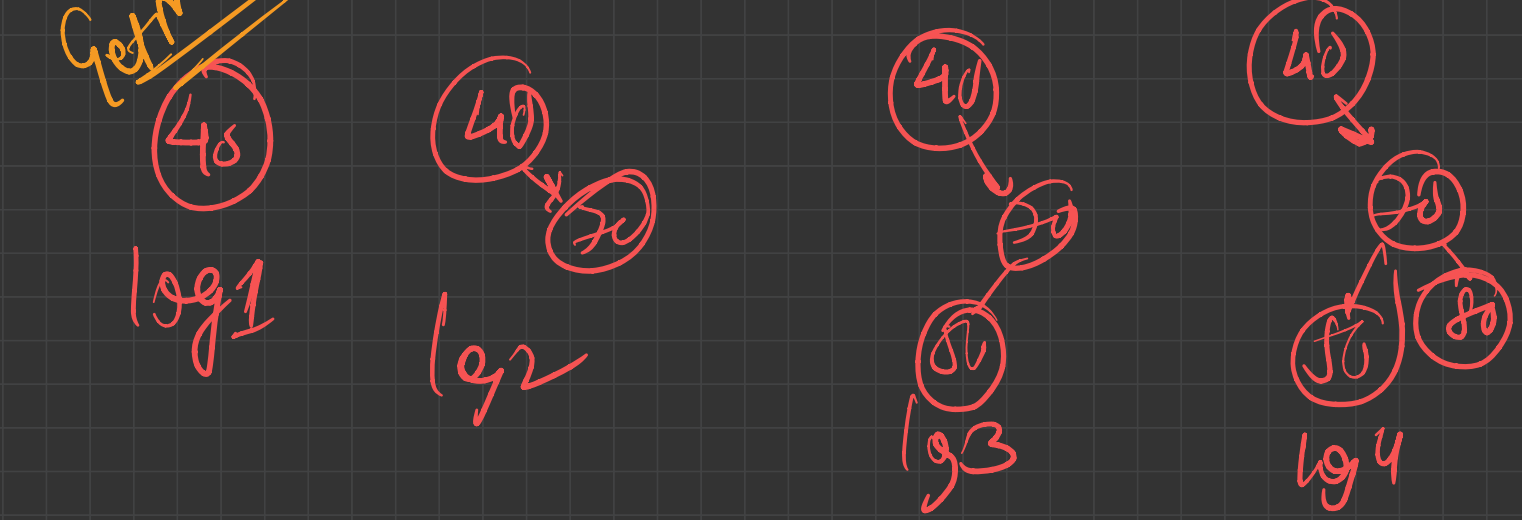
Approach ② Ordered_set / Treeset

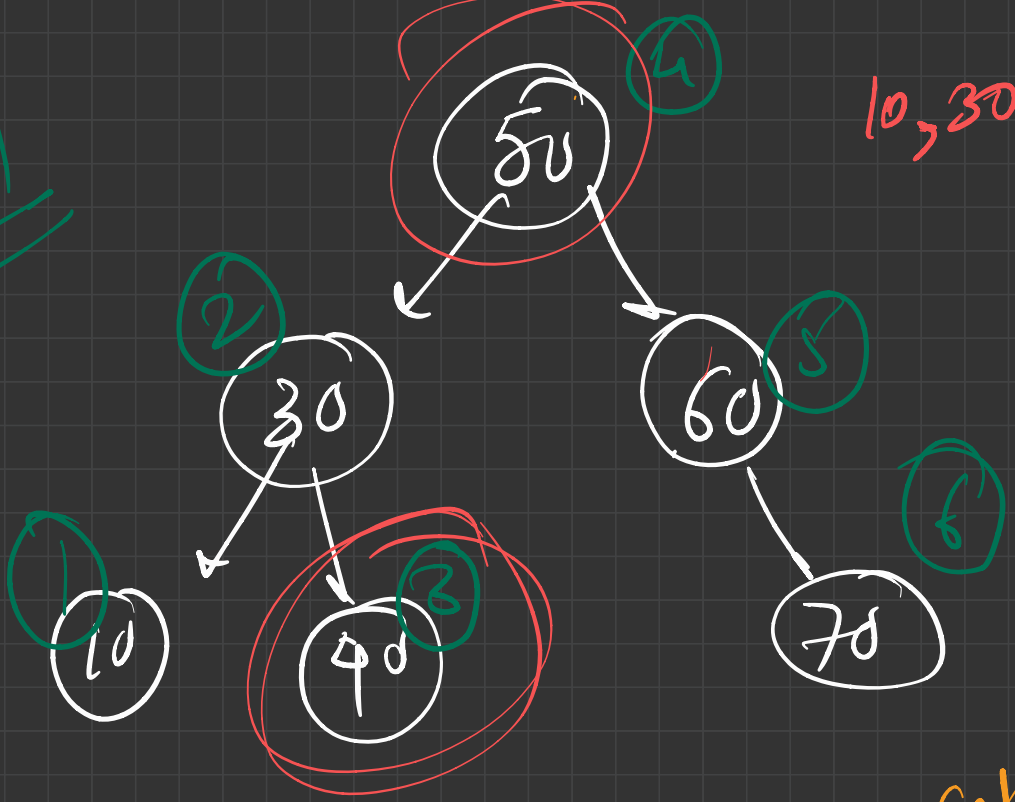
Binary search Tree

Insert → log n

Getmedian → O(1)

Sorted data





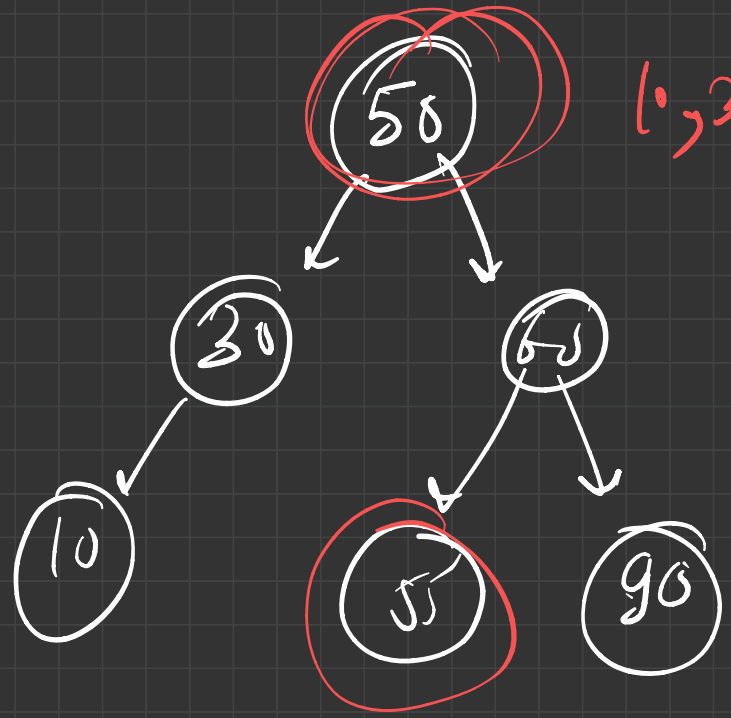
Q1

10, 30, 40, 50, 60, 70

inorder

getmedian

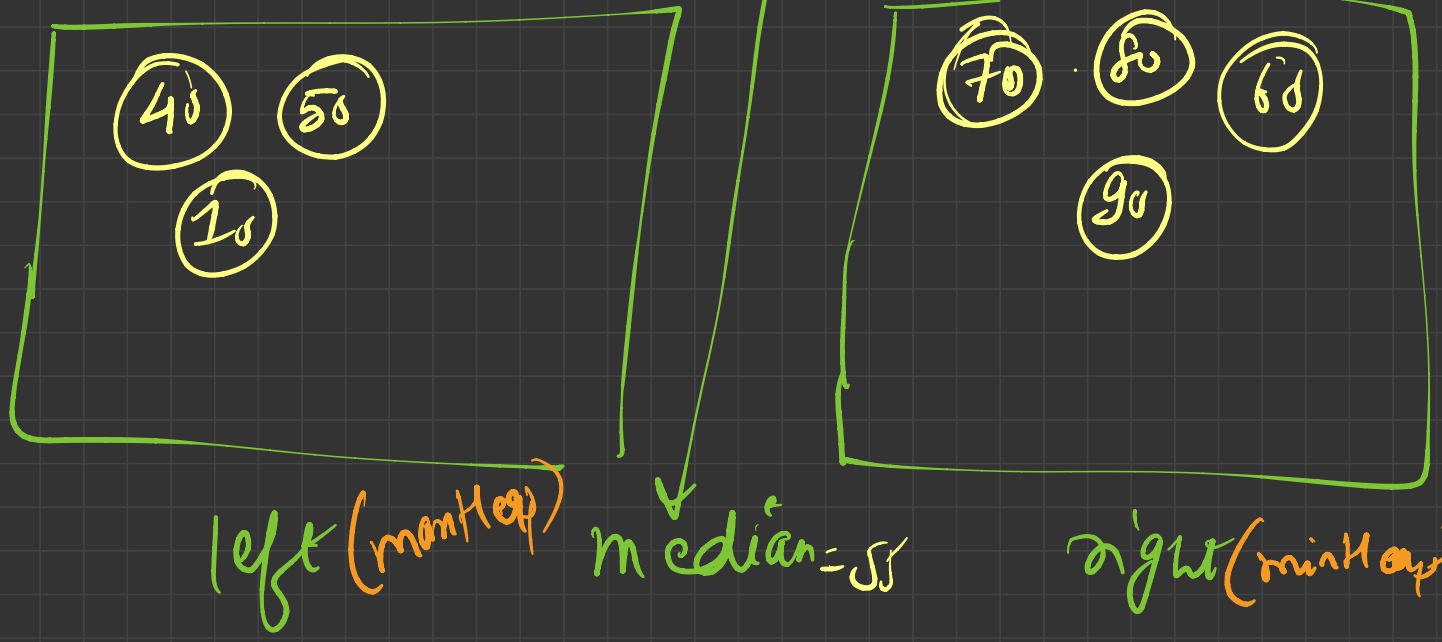
O(N)

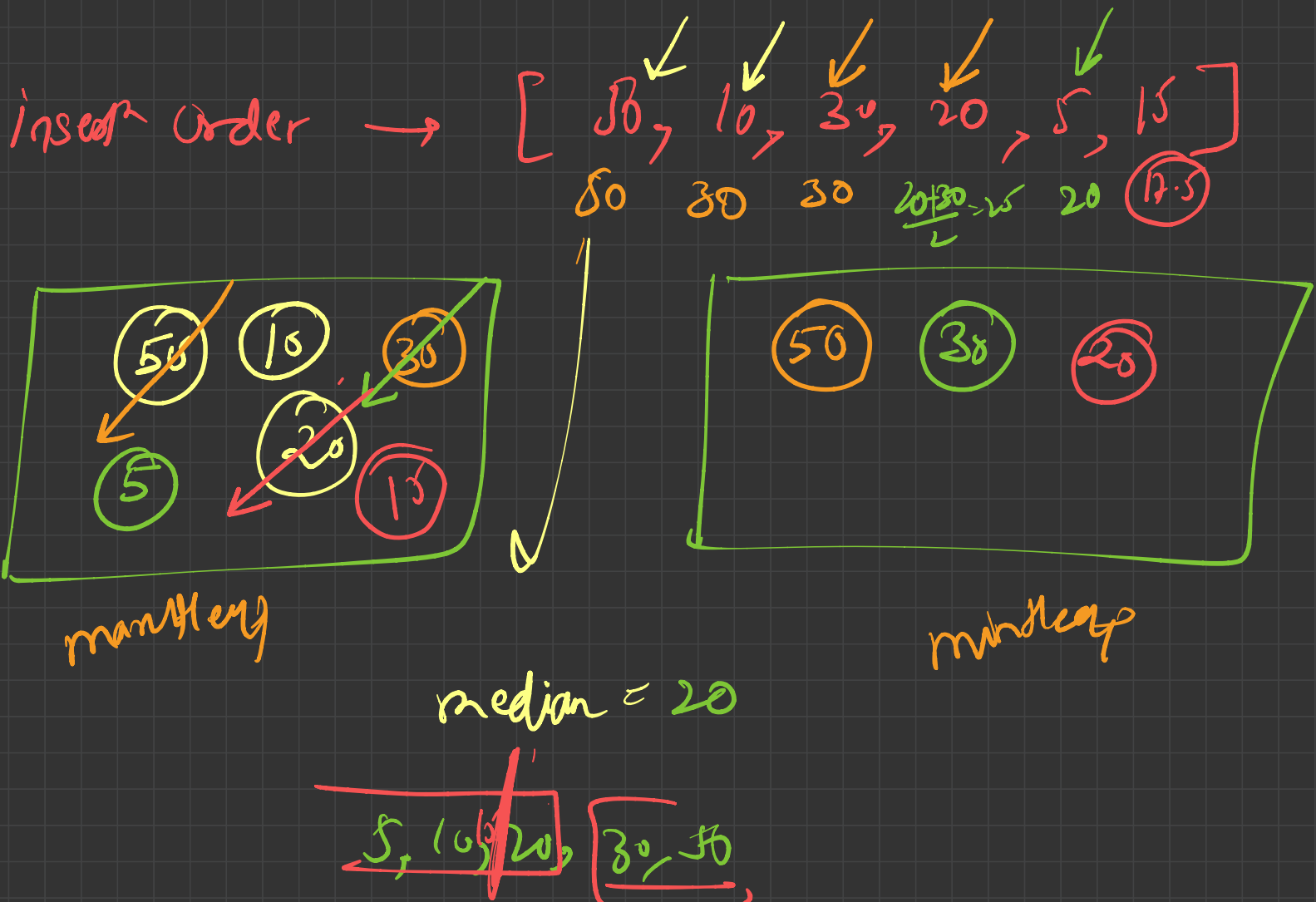

50
30
60
10
55
90

Approach ④ Priority queue

insert order → [40, 70, 50, 80, 10, 60, 90]

40 55 80 60 50 55 60




insert order →
[50, 10, 30, 20, 5, 15]
50 30 30 20+30=25 20 17.5
50
10
30
20
5
15
maxHeap
50
30
20
minHeap
median = 20
5, 10, 20, 30, 50

insertion order → [20, 50, 70, 60, 10, 30, 90]

20 35 50 55 50 40 50

max Heap (left)

min Heap (right)

10 : 55

```java
PriorityQueue<Integer> left, right;

public MedianFinder() {
    left = new PriorityQueue<>(Collections.reverseOrder()); // Max Heap
    right = new PriorityQueue<>(); // Min Heap
}
```

```java
public void addNum(int num) {
    // Insertion
    double median = findMedian();
    if(num <= median){
        left.add(num);
    } else {
        right.add(num);
    }

    // Balancing
    if(left.size() > right.size() + 1){
        right.add(left.remove());
    } else if(right.size() > left.size() + 1){
        left.add(right.remove());
    }
}
```

Insertion

Removal

```java
// Find Median -> Time = O(1)
public double findMedian() {
    if(left.size() + right.size() == 0) return -1;

    if(left.size() == right.size()){
        return (left.peek() + right.peek()) / 2.0;
    } else if(left.size() == right.size() + 1){
        return left.peek();
    } else {
        return right.peek();
    }
}
```

Insertion Time → (logN + logN)

Get median Time → O(1)

Recording → Backlogs ↑

Live classes

Reorganize String

"aaaaaabbcc"

↓

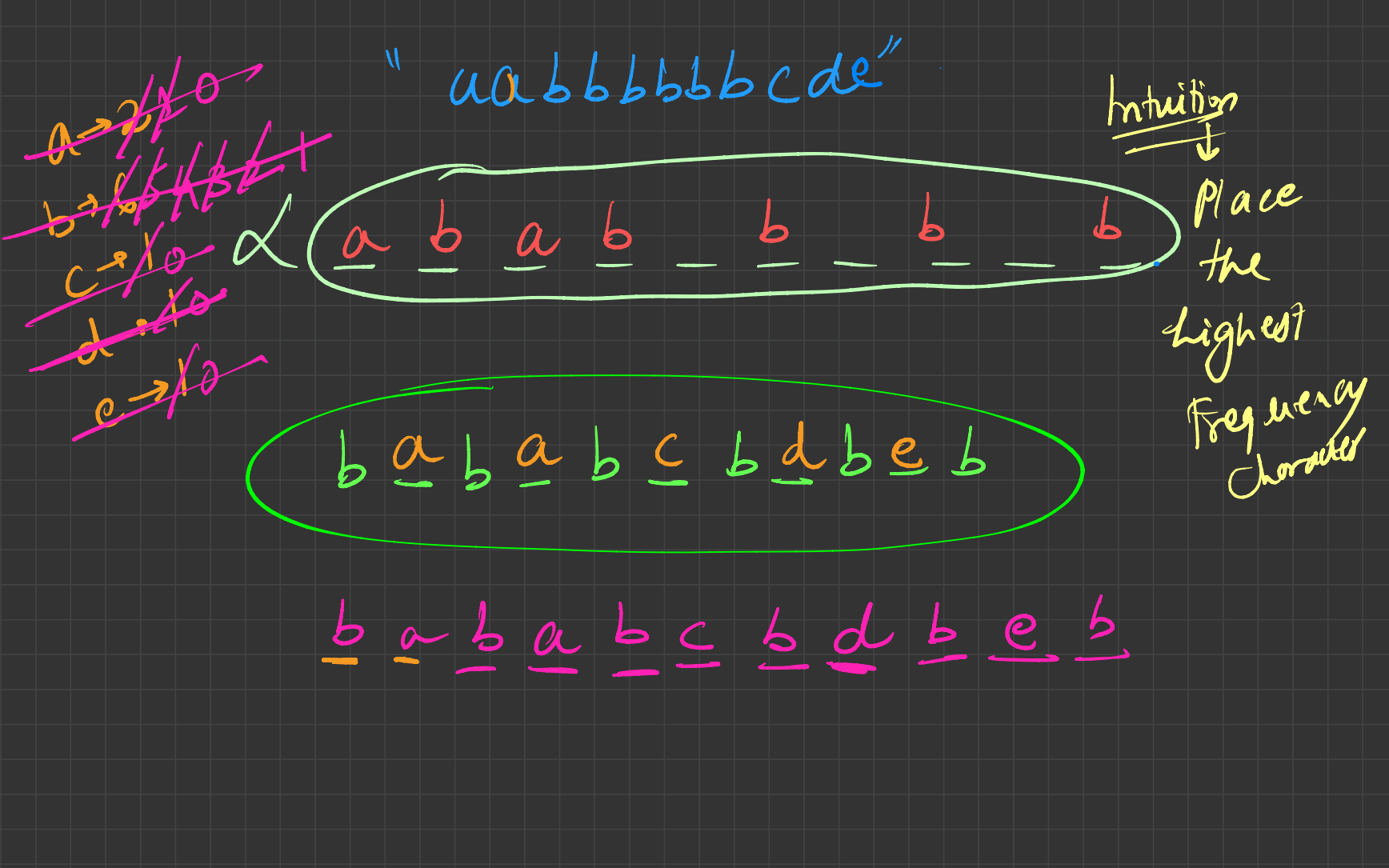
a b a b a b a c a c a

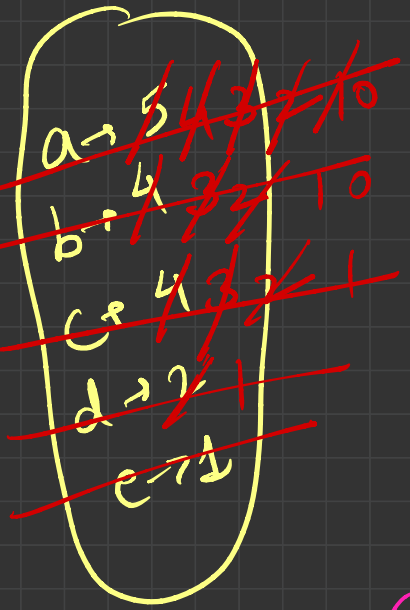
No two adjacent characters are same

"aaaabbbbcccdddd"

↓

a b a b a b a b c d c d c d c


"aabbbbbcde"
a → 2
b → 5
c → 1
d → 1
e → 1
a b a b _ b _ b _ b
b a b a b c b d b e b
b a b a b c b d b e b
Intuition
Place the highest frequency character



"a a a a a b b b b c c c c d d e"

5 4 4 2 1

a b a c a b c a b c d

a b c d e

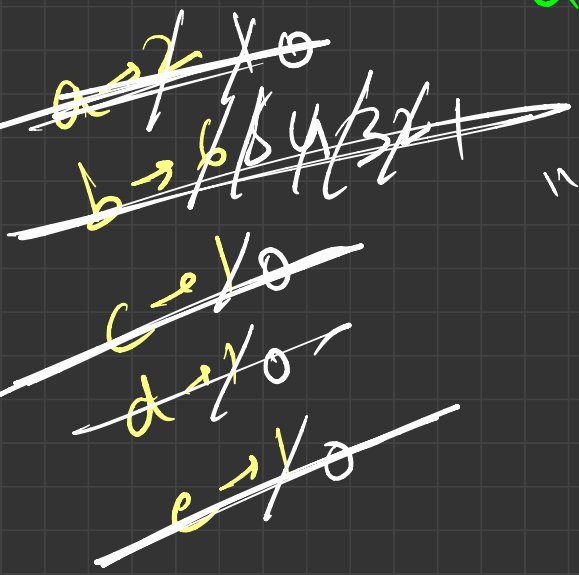
① Pop/Place the highest Frequency character first → max heap of pair

2 pairs

② Two pops cannot be of same character

pop → such that during 2nd pop, 1st pop is not in PQ

"aabbbbbcde"



"bababcbdbeb"

2pop → 2push alternatively

```java
public static class Pair{
    char ch;
    int freq;

    Pair(char ch, int freq){
        this.ch = ch;
        this.freq = freq;
    }
}

public static class FreqComp implements Comparator<Pair>{
    public int compare(Pair a, Pair b){
        // Higher Frequency -> Higher Priority
        return (b.freq - a.freq);
    }
}
```

max Heap
S2 - S1

Min Heap
S1 - S2

```java
HashMap<Character, Integer> freq = new HashMap<>();
for(int i=0; i<s.length(); i++){
    freq.put(s.charAt(i), freq.getOrDefault(s.charAt(i), 0) + 1);
}

PriorityQueue<Pair> q = new PriorityQueue<>(new FreqComp());
for(Character ch: freq.keySet()){
    q.add(new Pair(ch, freq.get(ch)));
}

StringBuilder str = new StringBuilder("");
```

```
StringBuilder str = new StringBuilder("");

while(q.size() > 0){
    Pair odd = q.remove();
    str.append(odd.ch);
    odd.freq--;

    if(q.size() > 0){
        Pair even = q.remove();
        str.append(even.ch);
        even.freq--;

        if(even.freq > 0){
            q.add(even);
        }
    }

    if(odd.freq > 0){
        q.add(odd);
    }
}

return str.toString();
```

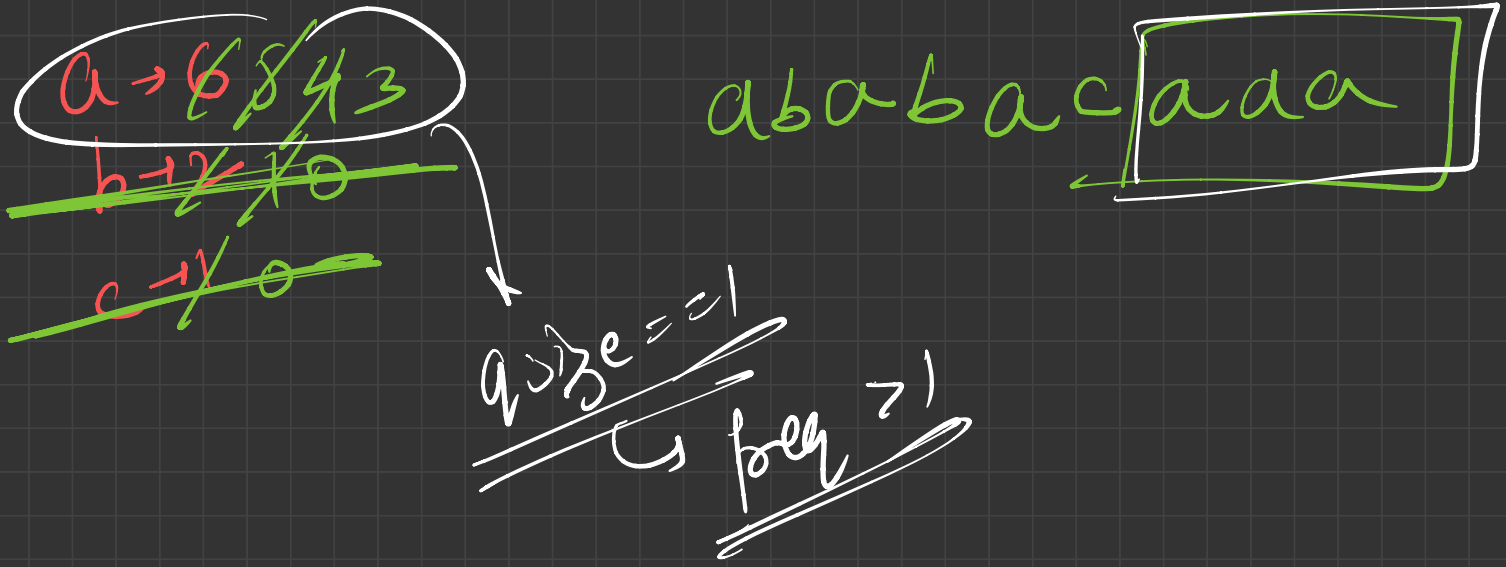
aaaa bbbb

a-4 3 2 1 0

b-4 3 2 1 0

abababab

"aaaaaaa bb c"



```
while(q.size() > 0){
    Pair odd = q.remove();
    str.append(odd.ch);
    odd.freq--;

    if(q.size() == 0 && odd.freq > 0){
        return "";
    }

    if(q.size() > 0){
        Pair even = q.remove();
        str.append(even.ch);
        even.freq--;

        if(even.freq > 0){
            q.add(even);
        }
    }

    if(odd.freq > 0){
        q.add(odd);
    }
}

return str.toString();
```