Heaps Notes:-

1️⃣ Kth Largest Element in a Stream (LeetCode)

Problem:

- Stream me naya number aata hai → return **current k-th largest** element

Heap type: Min Heap (size = k) → top() = kth largest

Approach:

1. Use **min heap**
2. Push all initial numbers → maintain size ≤ k
3. Add function → push new val → if size > k → pop → return top

```
class KthLargest {
    int k;
    priority_queue<int, vector<int>, greater<int>> pq;
public:
    KthLargest(int k, vector<int>& nums) {
        this->k = k;
        for (int num : nums) {
            pq.push(num);
            if (pq.size() > k) pq.pop();
        }
    }
    int add(int val) {
        pq.push(val);
        if (pq.size() > k) pq.pop();
        return pq.top();
    }
};
```

Dry Run:

- k = 3, nums = [4,5,8,2]

- Initial heap = [4,5,8] → top = 4
- Add 3 → heap = [4,5,8] → top = 4
- Add 5 → heap = [5,5,8] → top = 5

Time Complexity: O(log k) per add

Space Complexity: O(k)

Pattern / Template:

> Min Heap, size = k, top = kth largest

2️⃣ Merge K Sorted Lists (LeetCode)

Problem:

- k sorted linked lists → merge into one sorted list

Heap type: Min Heap (size ≤ k) → top = smallest current node

Approach:

1. Push **first node of each list** into min heap
2. Pop top → add to result
3. If node→next exists → push next into heap
4. Repeat until heap empty

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<pair<int,ListNode*>, vector<pair<int,ListNode*>>, greater
<pair<int,ListNode*>>> pq;
        for (auto node : lists) if (node) pq.push({node->val,node});
        ListNode* dummy = new ListNode(0);
        ListNode* tail = dummy;
        while (!pq.empty()) {
            auto top = pq.top(); pq.pop();
            tail->next = top.second; tail = tail->next;
            if (top.second->next) pq.push({top.second->next->val, top.second->ne
```

```
xt});
        }
        return dummy->next;
    }
};
```

Dry Run:

- Lists: [1→4→5, 1→3→4, 2→6]
- Heap initial: [1(L1),1(L2),2(L3)]
- Pop 1(L1) → push 4(L1) → Pop 1(L2) → push 3(L2) … → final merged list = 1→1->2→3→4→4→5→6

Time Complexity: O(N log k)

Space Complexity: O(k)

Pattern / Template:

> Min Heap, store {value,node pointer}, size ≤ k, push next nodes

3️⃣ Merge K Sorted Arrays

Problem:

- k sorted arrays → merge into one sorted array

Heap type: Min Heap (size ≤ k) → top = smallest current element

Approach:

1. Push first element of each array into heap → store {value, array_idx, element_idx}
2. Pop top → push value into result
3. If element_idx+1 exists → push next element into heap
4. Repeat until heap empty

```
vector<int> mergeKSortedArrays(vector<vector<int>>& arrays) {
    priority_queue<tuple<int,int,int>, vector<tuple<int,int,int>>, greater<tuple<i
```

```
nt,int,int>>> pq;
    vector<int> result;
    for (int i=0;i<arrays.size();i++)
        if (!arrays[i].empty()) pq.push({arrays[i][0],i,0});
    while (!pq.empty()) {
        auto [val,arrIdx,elemIdx] = pq.top(); pq.pop();
        result.push_back(val);
        if (elemIdx+1 < arrays[arrIdx].size())
            pq.push({arrays[arrIdx][elemIdx+1],arrIdx,elemIdx+1});
    }
    return result;
}
```

Dry Run:

- arrays = [[1,4,5],[1,3,4],[2,6]]
- Initial heap: [(1,0,0),(1,1,0),(2,2,0)] → pop 1, push next 4 → ... → final = [1,1,2,3,4,4,5,6]

Time Complexity: O(N log k)

Space Complexity: O(k)

Pattern / Template:

> Min Heap, store {value, array_idx, element_idx}, push next element

4️⃣ Task Scheduler (LeetCode)

Problem:

- Tasks = [A,A,A,B,B,B], cooling interval n
- Min total time to execute all tasks

Heap type: Max Heap (size ≤ 26) → top = most frequent task

Approach:

1. Count frequency of tasks
2. Push all frequencies into max heap

3. While heap not empty:
    - Loop i=0..n (cooling interval)
    - Pop top → execute → decrement frequency → if >0 store in temp
    - Increment time (idle if needed)
    - Push temp back into heap

```
int leastInterval(vector<char>& tasks, int n) {
    unordered_map<char,int> freq;
    for (char t: tasks) freq[t]++;
    priority_queue<int> pq;
    for (auto it: freq) pq.push(it.second);
    int time=0;
    while(!pq.empty()){
        vector<int> temp;
        int i=0;
        for(;i<=n;i++){
            if(!pq.empty()){
                int f=pq.top(); pq.pop(); f--;
                if(f>0) temp.push_back(f);
            }
            time++;
            if(pq.empty() && temp.empty()) break;
        }
        for(int f: temp) pq.push(f);
    }
    return time;
}
```

Dry Run:

- tasks = [A,A,A,B,B,B], n=2
- freq: A=3,B=3 → heap: [3,3]
- Interval 1: pop A,B → push back [2,2] → time += 3 (idle included)

- Interval 2: pop A,B → push back [1,1] → time += 3
- Interval 3: pop A,B → heap empty → time +=2
- Total time = 8

Time Complexity: O(N)

Space Complexity: O(26)

Pattern / Template:

> Max Heap → top = most frequent task, loop n+1 units, push remaining back