**Implementation of Heaps**

There are 3 functions in heap

1. Heapify
2. ExtractMAX or ExtractMIN
3. Insert

1 Heapify -

* Check for the children of I
* If present then compare the children with the parent.
* If the parent is max then return
* Swap the parent with the greatest child.
* Move the iterator to that child and do the same till a parent with no child is found.

2 ExtractMax -

* Swap the first and the last element of the heap.
* Apply heapify on the first element.

3 Insert -

* Insert the element in the end of the list.
* Compare that element with its parent

Note – If i is a parent then 2i+1 and 2i+2 are children. If i is child then (i-1)/2 is the parent.

import java.util.\*;

public class Heaps{

public void heapify(ArrayList<Integer> heap, int i) {

int n = heap.size();

while(2\*i+1 < n) {

int mx = Math.max(heap.get(i), heap.get(2\*i+1));

if(2\*i+2 < n) mx = Math.max(mx, heap.get(2\*i+2));

if(mx == heap.get(i)) return;

else if(mx == heap.get(2\*i+1)) {

Collections.swap(heap, i, 2\*i+1);

i = 2\*i+1;

} else {

if(2\*i+2 >= n) return;

Collections.swap(heap, i, 2\*i+2);

i = 2\*i+2;

}

}

}

public int extractMax(ArrayList<Integer> heap) {

int mx = heap.get(0);

int n = heap.size();

Collections.swap(heap, 0, n-1);

heap.remove(n-1);

heapify(heap, 0);

return mx;

}

public void insert(ArrayList<Integer> heap, int x) {

heap.add(x);

int n = heap.size()-1;

while(n!=0) {

int par = (n-1)/2;

if(heap.get(par) > heap.get(n)) return;

else {

Collections.swap(heap, par, n);

n = par;

}

}

}

public int poll(ArrayList<Integer> heap) {

return isEmpty(heap) ? heap.get(0) : -1;

}

public boolean isEmpty(ArrayList<Integer> heap) {

return heap.size() > 0;

}

}

**Q1. B-th Smallest Prime Fraction**

Given a sorted array of integers **A** which contains **1** and some number of primes.  
Then, for every **p < q** in the list, we consider the fraction **p / q**.

What is the **B-th smallest** fraction considered?

Return your answer as an array of integers, where **answer[0] = p** and **answer[1] = q**.

**Logic –**

We can add all possible fractions in heap and remove smallest B elements.

But this can take a lot of time.

We add the smallest possible fraction using each element in the heap.

Example => [1 2 3 5] we add 1/5, 2/5, 3/5 in heap.

Later on, we remove the minimum one from heap and insert the next smallest fraction for that element in the heap. Eg => we remove 1/5 and add ¼ in the heap.

**Code -**

public class Solution {

public class Node{

double data;

int p;

int q;

public Node(int a,int b, double d) {

p = a;

q = b;

data = d;

}

}

public class myCMP implements Comparator<Node> {

public int compare(Node a, Node b) {

if(a.data > b.data) return 1;

else if(b.data > a.data) return -1;

return 0;

}

}

public ArrayList<Integer> solve(ArrayList<Integer> A, int B) {

PriorityQueue<Node> pq = new PriorityQueue<Node>(new myCMP());

int n = A.size();

for(int i=0;i<n-1;i++) {

pq.add(new Node(i, n-1, Float.valueOf(A.get(i))/Float.valueOf(A.get(n-1))));

}

Node x = null;

for(int i=0;i<B;i++) {

x = pq.poll();

pq.add(new Node(x.p, x.q-1,Float.valueOf(A.get(x.p))/Float.valueOf(A.get(x.q-1))));

}

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

ans.add(A.get(x.p));

ans.add(A.get(x.q));

return ans;

}

}

**Q2. Running Median**

Given an array of integers, **A** denoting a stream of integers. New arrays of integer **B** and **C** are formed.  
Each time an integer is encountered in a stream, append it at the end of **B** and append the median of array **B** at the **C**.

Find and return the array **C**.

**NOTE:**

* If the number of elements is **N** in **B** and **N** is odd, then consider the median as **B[N/2]** ( B must be in sorted order).
* If the number of elements is **N** in **B** and **N** is even, then consider the median as **B[N/2-1].** ( B must be in sorted order).

**Logic -**

1. Create two heaps. One max heap (A) to maintain elements of the lower half and one min heap(B) to maintain elements of the higher half at any point in time.
2. If incoming element is greater than A.peek() then add in B else add in A.
3. Then, we have to balance the heaps such that if there are even number of elements then both heaps should have same number of elements else if there are odd number of elements heap A should have 1 element extra.

**Code -**

public class Solution {

public ArrayList<Integer> solve(ArrayList<Integer> arr) {

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

PriorityQueue<Integer> B = new PriorityQueue<Integer>();

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

int sizeA = 0;

int sizeB = 0;

for(int i=0;i<arr.size();i++) {

if(A.isEmpty() || arr.get(i) > A.peek()) {

B.add(arr.get(i));

sizeB += 1;

}

else if(arr.get(i) <= A.peek()) {

A.add(arr.get(i));

sizeA += 1;

}

if(sizeB > sizeA) {

A.add(B.poll());

sizeB -= 1;

sizeA += 1;

} else {

if(sizeA - sizeB > 1) {

B.add(A.poll());

sizeA -= 1;

sizeB += 1;

}

}

ans.add(A.peek());

}

return ans;

}

}

**Q3. K Places Apart**

**N** people having different priorities are standing in a queue.

The queue follows the **property** that each person is standing at most **B** places away from its position in the sorted queue.

Your **task** is to sort the queue in the increasing order of priorities.

**NOTE:**

* No two persons can have the same priority.
* Use the property of the queue to sort the queue with complexity O(NlogB).

**Logic -**

Take the first B+1 elements in a list, then finding the minimum of the list will give us the first element in the sorted array.

Remove the smallest element from the list and add the (B+2)th element from the given array to the list.

Now taking the minimum of the list will give us the second element in the sorted array as each element is <= B distance away from its sorted position.

Proceed in the same way and keep on finding the minimum of B+1 elements.

We can use priority\_queue to implement the above solution. Each time we remove the minimum element from the queue and add the new element to the queue.

**Time complexity** of removing an element from the priority\_queue will be O(log B) as the size of the queue is <= B+1, and we will remove all N elements from the queue one by one. So, overall time complexity will be O(NlogB).

public class Solution {

public ArrayList<Integer> solve(ArrayList<Integer> A, int B) {

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

for(int i=0;i<B;i++) {

pq.add(A.get(i));

}

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

for(int i=B;i<A.size();i++) {

pq.add(A.get(i));

ans.add(pq.poll());

}

while(!pq.isEmpty()) {

ans.add(pq.poll());

}

return ans; } }

**Q4. Ath largest element**

Given an integer array **B** of size **N**.

You need to find the **Ath** largest element in the **subarray [1 to i],** where **i** varies from **1** to **N**. In other words, find the **Ath** largest element in the sub-arrays **[1 : 1], [1 : 2], [1 : 3], ...., [1 : N]**.

**NOTE:** If any subarray [1 : i] has less than A elements, then the output should be **-1** at the **ith** index.

**Logic -**

Compare the new number value X with the peek of the min heap. If X is smaller, then ignore it.  
Otherwise, extract the minimum element in the heap and add the new number X.

The time complexity of adding a new element and finding the Kth highest element is O(logK).  
STL priority queue can be used to implement a heap directly.

**Code -**

public class Solution {

public ArrayList<Integer> solve(int A, ArrayList<Integer> B) {

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

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

for(int i=0;i<A-1;i++) {

ans.add(-1);

pq.add(B.get(i));

}

pq.add(B.get(A-1));

ans.add(pq.peek());

for(int i=A;i<B.size();i++) {

if(B.get(i) > pq.peek()) {

pq.poll();

pq.add(B.get(i));

}

ans.add(pq.peek());

}

return ans;

}

}

**Q5. Minimum largest element**

Given an array **A** of **N** numbers, you have to perform **B** operations. In each operation, you have to pick any one of the **N** elements and add the original value(value stored at the index before we did any operations) to its current value. You can choose any of the **N** elements in each operation.

Perform **B** operations in such a way that the largest element of the modified array(after **B** operations) is minimized.  
Find the minimum possible largest element after **B** operations.

**Logic -**

1. Create a class which will store old and new value of the elements in array.
2. Add all elements in the priority queue
3. Extract the minimum element with minimum old\_val + new\_val
4. Then, add the old value in the new value and add back the node to Priority queue

**Code -**

public class Solution {

public class Node {

int old\_val;

int new\_val;

public Node(int a, int b) {

old\_val = a;

new\_val = b;

}

}

public class myCMP implements Comparator<Node> {

public int compare(Node a, Node b) {

return (a.old\_val+a.new\_val - b.old\_val-b.new\_val);

}

}

public int solve(ArrayList<Integer> A, int B) {

PriorityQueue<Node> pq = new PriorityQueue<Node>(new myCMP());

for (int i=0;i<A.size();i++) pq.add(new Node(A.get(i), A.get(i)));

for(int i=0;i<B;i++) {

Node x = pq.poll();

pq.add(new Node(x.old\_val, x.new\_val+x.old\_val));

}

int ans = Integer.MIN\_VALUE;

while(!pq.isEmpty()) {

Node x = pq.poll();

ans = Math.max(x.new\_val, ans);

}

return ans;

}

}

**Q6. N max pair combinations**

Given two integers arrays, **A** and **B,** of size **N** each.

Find the **maximum N elements** from the sum combinations (Ai + Bj) formed from elements in arrays A and B.

**Logic -**

1. Sort the array in desc order.
2. Create a class which holds the indexs i, j and the sum of A[i] and B[j]
3. Keeping one index i = 0 add the indexs of the second array.
4. Then remove the node with largest sum, and add the node with the same j but with i = i+1.

**Code -**

public class Solution {

public class Node {

int Ai;

int Aj;

int sum;

public Node(int i,int j, ArrayList<Integer> arri, ArrayList<Integer> arrj) {

Ai = i;

Aj = j;

sum = arri.get(Ai) + arrj.get(Aj);

}

}

public class myCMP implements Comparator<Node> {

public int compare(Node a, Node b) {

return b.sum - a.sum;

}

}

public ArrayList<Integer> solve(ArrayList<Integer> A, ArrayList<Integer> B) {

Collections.sort(A, Collections.reverseOrder());

Collections.sort(B, Collections.reverseOrder());

PriorityQueue<Node> pq = new PriorityQueue<Node>(new myCMP());

for(int i=0;i<A.size();i++) pq.add(new Node(i, 0, A, B));

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

for(int i=0;i<A.size();i++) {

Node x = pq.poll();

ans.add(x.sum);

if(x.Aj < A.size()-1) pq.add(new Node(x.Ai, x.Aj+1, A, B));

}

return ans;

}

}

**Q7. Special Median**

You are given an array **A** containing **N** numbers. This array is called **special** if it satisfies one of the following properties:

1. There exists an element A[i] in the array such that A[i] is equal to the median of elements [A[0], A[1], ...., A[i-1]]
2. There exists an element A[i] in the array such that A[i] is equal to the median of elements [A[i+1], A[i+2], ...., A[N-1]]

The **Median** is the **middle** element in the sorted list of elements. If the number of elements is **even** then the median will be (sum of both middle elements) / 2.

Return **1** if the array is special else return **0**.

**NOTE:**

* Do not neglect decimal point while calculating the median
* For A[0] consider only the median of elements [A[1], A[2], ..., A[N-1]] (as there are no elements to the left of it)
* For A[N-1] consider only the median of elements [A[0], A[1], ...., A[N-2]]

**Logic -**

1. Maintain a maxheap and a minheap.
2. Find the medians in 2 directions. That is traverse from left to right and then from right to left.
3. Write a function adds and maintains a balance between the 2 heaps.
4. Write a function to find the median till [0 to i].
5. Int the left to right traversal compare A[i+1] with current median, if they are equal return 1.
6. In the right to left compare A[i-1] with current median, if they are equal return 1.

**Code -**

public class Solution {

static PriorityQueue < Integer > maxHeap;

static PriorityQueue < Integer > minHeap;

public int solve(int[] A) {

int n = A.length;

minHeap = new PriorityQueue();

maxHeap = new PriorityQueue(new CustomComp());

for (int i = 0; i < (n - 1); i++) {

add\_number(A[i]);

if (A[i + 1] == find\_median()) return 1;

}

minHeap = new PriorityQueue();

maxHeap = new PriorityQueue(new CustomComp());

for (int i = n - 1; i > 0; i--) {

add\_number(A[i]);

if (A[i - 1] == find\_median()) return 1;

}

return 0;

}

public static void add\_number(int num) {

maxHeap.offer(num);

minHeap.offer(maxHeap.peek());

maxHeap.poll();

if (maxHeap.size() < minHeap.size()) {

maxHeap.offer(minHeap.peek());

minHeap.poll();

}

}

public static int find\_median() {

if (maxHeap.size() > minHeap.size()) return maxHeap.peek();

else {

int ans = maxHeap.peek() + minHeap.peek();

if (ans % 2 != 0) return Integer.MAX\_VALUE; //answer is a fraction

else return ans / 2;

}

}

}

class CustomComp implements Comparator < Integer > {

@Override

public int compare(Integer a, Integer b) {

return b - a;

}

}