Revision - Heaps

Time Complexities of Some Common Data Structures

	Insert(X)	getMin()	deleteMin()
Sorted Array	O(N)	O(1)	O(N) - ASC, O(1) - DESC
Linked List (Sorted)	O(N)	O(1)	O(1)
Binary Search Tree	O(N)	O(N)	O(N)
Balanced Binary Search Tree	O(logN)	O(logN)	O(logN)
Min Heap	O(logN)	O(1)	O(logN)

In a Min or Max heap we try to maintain a complete binary tree with minimum or maximum element as the root of the tree.

Properties of Heap

1. It is a complete binary tree.
2. For all nodes in a min heap, node.value <= node.left.value and node.value <= node.right.value. Simila logic can be used for max heap.

Suppose we have array elements A = [2, 3, 1, 6, 4]
A min heap would look like:

```
      1
     / \
     2   3
   / \
  4    6
```

Corresponding array representation would be:
T = [1, 2, 3, 4, 6]

Insertion in Min Heap

1. Insert the new element by creating a new leaf.
2. Keep on swapping it with the parent until the parent node is smaller. This is a down heapify operation.

Pseudo Code:

```
function(T, K):
    N = length of T
    append K to T
    cur = N
    while cur > 0:
        parent = (cur - 1)/2
        if(T[parent] < T[cur]):
            swap(T[parent], T[cur])
        else:
            break loop
        cur = parent

    return T
```

Same logic is used for max heaps.

Deletion Minimum in Min Heap

1. Swap the minimum element with the last element of the min heap.
2. Now, climb down from root and heapify it. This is an up heapify operation.

Pseudo Code:

```
function(T):
    N = length of T
    swap(T[0], T[N - 1])
    last = N - 2
    i = 0
    while i < last:
        minIndex = i
        l = 2 * i + 1
        r = 2 * i + 2
        if l <= last and A[l] < A[minIndex]:
            minIndex = l
        if r <= last and A[r] < A[minIndex]:
            minIndex = r
        if minIndex == i:
            break
        swap(A[i], A[minIndex])
        i = minIndex
```

Same logic is used for max heaps.

Question - 1

Given a 2D matrix of size NxM with sorted rows. Merge all the rows into a sorted 1D array.

Example:
[[1, 3, 8],
[2, 5, 7],
[4, 9, 12]]

Here, the respective 1D array would be:
[1, 2, 3, 4, 5, 7, 8, 9, 12]

Solution:

Brute force:

- Insert all the elements in an array.
- Sort the array using any sorting algorithm
- Time complexity: O(N * M * log(N * M))

Pseudo Code:

```
function(A):
    rows = length of A
    cols = length of A[0]
    ans = []
    for i from 0 to rows - 1:
        for j from 0 to cols - 1:
            append A[i][j] to ans

    sort(ans)
    return ans
```

Observation:
We observe that the matrix is sorted. So, we can use this property to efficiently create the sorted 1D array.

Pseudo Code:

```
Data:
    val, r, c

function(A):
    N = length of A
    M = length of A[0]
    H = Heap()
    for i from 0 to N - 1:
        insert(Data(A[i][0], i, 0)) in H

    ans = []
    while H is not empty:
        T = Min of H
        remove T from H
        append T.val to ans
        if T.c < M - 1:
            insert(Data(A[T.r][T.c + 1], T.r, T.c + 1)) in H

    return ans
```

- Time complexity: O(N * M * log(N))
- Space complexity: O(N)

Question - 2

Given two sorted arrays A and B, print K^{th} smallest element.

Example:
A = [1, 2, 3]
B = [2, 4, 5]
K = 2

All possible pairs are:
[3, 4, 5, 6, 7, 8]

2^{nd} smallest is 4.

Solution:

Brute force:
Simply create the array of all possible pair sums and sort them.

Pseudo Code:

```
function(A, B, K):
    C = []
    for a in A:
        for b in B:
            append (a + b) in C

    sort(C)
    return C[K - 1]
```

- Time complexity: O(N * M * log(N * M))
- Space complexity: O(N * M)

Observation:
If we get an element (i, j) such that its rank < K, then our possible answers can be (i + 1, j) or (i, j + 1)

Pseudo Code:

```
function(A, B, K):
    N = length(A)
    M = length(B)
    i = 0
    j = 0
    H = Heap()
    rank = 0
    ans = -1
    while(rank < K):
        rank++
        T = min of H
        remove T from H
        add(A[i + 1] + B[j]) to H
        add(A[i] + B[j + 1]) to H

    return ans
```

- Time complexity: O(K * log(K))
- Space complexity: O(K)

Question - 3

You are getting a stream of integers. With every new element you get, return the current median.

Example:
Suppose the stream is: [4, 6, 3, 2, 9]
After 1st element: [4] -> Median = 4

After 2nd element: [4, 6] -> Median = 5
After 3rd element: [4, 6, 3] -> [3, 4, 6] -> Median = 4
After 4th element: [4, 6, 3, 2] -> [2, 3, 4, 6] -> Median = 3.5
After 5th element: [4, 6, 3, 2, 9] -> [2, 3, 4, 6, 9] -> Median = 4

Solution:

Brute force:
Whenever we receive the new element, simply add it to an array. After which, we sort the array to find the median elements.

- Time complexity: $O(N^2 \log(N))$, as we are sorting the array after every insertion of the element.
- Space complexity: O(N)

Optimization:
We can observe from the brute force solution is that what we care about after every insertion is the middle elements. We can use a pair of min and max heap to maintain this information in an efficient manner.

Pseudo Code:

```
function(A):
    minHeap = MinHeap()
    maxHeap = MaxHeap()
    added = 0

    for ele in A:
        added = added + 1
        add ele in maxHeap
        if maxHeap.max > minHeap.min:
            max = maxHeap.max
            min = minHeap.min
            remove max from maxHeap
            remove min from minHeap
            add max to minHeap
            add min to maxHeap
        if size(maxHeap) > size(minHeap) + 1:
            max = top of maxHeap
            remove max from maxHeap
            add max to minHeap
        median = -1
        if added is odd:
            median = maxHeap.max
        else:
            median = (maxHeap.max + minHeap.min)/2
        print(median)
```

- Time complexity: O(log(N)) per insertion, so O(N * log(N)) for all insertions

- Space complexity: O(N)

Heap Sort

In this sorting algorithm, we simply convert the unsorted array into a min heap. After this, we continously extract minimum element from the heap and create a new sorted array.

- Time complexity: O(N) + O(N * log(N))
- Space complexity: O(1) -> We can use in place sorting to attain this time complexity.

Pseudo Code:

```
function(A):
    minHeap = MinHeap(A) // in O(N)
    ans = []

    // O(N * log(N))
    while minHeap is not empty:
        min = min element of minHeap
        append min to ans
    return ans
```