**Q1. Task Scheduling**

A CPU has **N** tasks to be performed. It is to be noted that the tasks have to be completed in a specific order to avoid deadlock. In every clock cycle, the CPU can either **perform a task or move it to the back of the queue**. You are given the current state of the scheduler queue in array **A** and the required order of the tasks in array **B**.  
  
Determine the minimum number of clock cycles to complete all the tasks.

**Logic -**

1. Enter all elements of the array A in queue.
2. Remove the element and add it to the end of the queue till B[i] == q.peek() (inc cost).
3. Remove the element from queue where q.peek() == B[i] (inc cost).

**Code** -

public class Solution {

public int solve(int[] A, int[] B) {

Queue<Integer> q = new LinkedList<Integer>();

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

q.add(A[i]);

}

int cost = 0;

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

while(B[i] != q.peek()) {

q.add(q.poll());

cost+=1;

}

q.remove();

cost+=1;

}

return cost;

}

}

**Q2: N integers containing only 1, 2 & 3**

Given an integer, **A**. Find and Return first positive **A** integers in ascending order containing only digits 1, 2, and 3.

**NOTE:** All the **A** integers will fit in 32-bit integers.

**Logic –** Simple BFS type solution

1. We know the initial three values will be 1, 2, and 3.
2. Now, the upcoming values will be by appending 1, 2, and 3 in each given value.
3. We will use a queue to store the elements in ascending order.

**Code -**

public class Solution {

public int[] solve(int A) {

int [] ans = new int[A];

ans[0] = 1;

if (A==1) return ans;

ans[1] = 2;

if (A==2) return ans;

ans[2] = 3;

if (A==3) return ans;

Queue<Integer> q = new LinkedList<>();

q.add(1);

q.add(2);

q.add(3);

int i = 3;

int temp2 = 10;

while(i<A) {

int temp = 10 \* q.peek() + 1;

ans[i] = temp;

q.add(temp);

i+=1;

if(i==A) return ans;

temp = 10 \* q.peek() + 2;

ans[i] = temp;

q.add(temp);

i+=1;

if(i==A) return ans;

temp = 10 \* q.peek() + 3;

ans[i] = temp;

q.add(temp);

i+=1;

q.remove();

}

return ans;

}

}

**Q3. First non-repeating character**

Given a string **A** denoting a stream of lowercase alphabets, you have to make a new string B.  
B is formed such that we have to find the first non-repeating character each time a character is inserted to the stream and append it at the end to B. If no non-repeating character is found, append '#' at the end of B.

**Logic -**

1. Maintain a queue and a hashmap which stores frequency of characters.
2. Traverse the String and perform below steps.
   1. Update the frequency for the character in the hashmap.
   2. Insert the element in the Queue (from the rear end).
   3. Remove the useless elements (appearing more than once) from the front end of the queue.
   4. Append the front value to the array.

**Code -**

public class Solution {

public String solve(String A) {

StringBuilder ans = new StringBuilder();

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

Queue<Character> queue = new LinkedList<Character>();

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

// Insert in map

if(!map.containsKey(A.charAt(i))) {

map.put(A.charAt(i), 0);

}

map.put(A.charAt(i), map.get(A.charAt(i))+1);

// Insert in queue

queue.add(A.charAt(i));

// Remove useless elements from queue

while(queue.size() > 0 && map.get(queue.peek()) > 1) {

queue.remove();

}

if (queue.size() > 0) ans.append(queue.peek());

else ans.append('#');

}

return ans.toString();

}

}

**Q4. Sum of min and max**

Given an array **A** of both positive and negative integers.

Your task is to compute the sum of **minimum** and **maximum** elements of all sub-array of size **B**.

**NOTE:** Since the answer can be very large, you are required to return the sum modulo 109 + 7.

**Logic -**

1. We will use Deque(Double-Ended Queue) and the concept of the sliding window.
2. We create two empty double-ended queues of size B (‘S’ , ‘G’) that only store indexes of elements of the current window that are not useless.  
   An element is useless if it can not be the maximum or minimum of the next subarrays.  
   -> In deque ‘G’, we maintain decreasing order of values from front to rear.  
   -> In deque ‘S’, we maintain increasing order of values from front to rear.
3. Maintain both Dequeue such that the front element gives maximum and minimum element respectively for each window.  
   Add that element to the sum variable.  
   Return the sum%10^9+7.  
   Note that the sum%10^9+7 will be in the range (0,10^9+6).

**Code -**

public class Solution {

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

Deque<Integer> minq = new ArrayDeque < Integer > ();

Deque<Integer> maxq = new ArrayDeque < Integer > ();

int mod = 1000 \* 1000 \* 1000 + 7;

// First Window

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

while(!minq.isEmpty() && minq.getLast() > A.get(i)) {

minq.removeLast();

}

minq.addLast(A.get(i));

while(!maxq.isEmpty() && maxq.getLast() < A.get(i)) {

maxq.removeLast();

}

maxq.addLast(A.get(i));

}

long ans += maxq.getFirst() + minq.getFirst();

// Sliding

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

int incoming = A.get(i+B);

int outgoing = A.get(i);

while(!maxq.isEmpty() && maxq.getLast() < incoming) {

maxq.removeLast();

}

maxq.add(incoming);

if(maxq.peek() == outgoing) maxq.removeFirst();

while(!minq.isEmpty() && minq.getLast() > incoming) {

minq.removeLast();

}

minq.add(incoming);

if(minq.peek() == outgoing) minq.removeFirst();

ans += minq.getFirst() + maxq.getFirst();

ans %= mod;

}

ans += mod;

ans %= mod;

return (int)(ans % (1000\*1000\*1000+7));

}

}

**Q5. Maximum Frequency stack**

You are given a matrix **A** of size N x 2 which represents different operations.  
Assume initially you have a stack-like data structure you have to perform operations on it.

Operations are of two types:

1 x: push an integer x onto the stack and return -1.

2 0: remove and return the most frequent element in the stack.

If there is a tie for the most frequent element, the element closest to the top of the stack is removed and returned.

A[i][0] describes the type of operation to be performed. A[i][1] describe the element x or 0 corresponding to the operation performed.

**Logic -**

1. Maintain a frequency hashmap (map1) for storing frequency of the numbers.
2. Maintain a hashmap(map2) which will store the freq as keys and a linked list as a value.
3. When we push an element we will push the element in the linked list linked to the frequency of that element (freq = map1.get(element) + 1)

**Code -**

public class Solution {

public class Node {

int val;

Node next;

public Node(int d) {

val = d;

next = null;

}

}

public int[] solve(int[][] A) {

HashMap<Integer, Integer> map = new HashMap<>();

HashMap<Integer, Node> stackMap = new HashMap<>();

int[] ans = new int[A.length];

int maxFreq = 0;

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

// Push

if (A[i][0] == 1) {

int temp = A[i][1];

// Add temp in HashMap

if(!map.containsKey(temp)) {

map.put(temp, 0);

}

map.put(temp, map.get(temp)+1);

// Update maxFreq

int freq = map.get(temp);

maxFreq = Math.max(freq, maxFreq);

// Check freq in stackMap

// If not present create new linked list and put it in stackMap

if(!stackMap.containsKey(freq)) {

Node st = new Node(temp);

stackMap.put(freq, st);

// Else insert at head of the linked list at stackMap.get(maxFreq)

} else {

Node st = stackMap.get(freq);

Node newNode = new Node(temp);

newNode.next = st;

st = newNode;

stackMap.put(freq, st);

}

ans[i] = -1;

}

// Pop

else {

// Get the linked list at maxFreq

Node st = stackMap.get(maxFreq);

// Reduce freq of the element to be popped

int z = st.val;

map.put(z, map.get(z)-1);

// Remove element from the linked list at maxFreq

ans[i] = st.val;

st = st.next;

if (st==null) {

stackMap.remove(maxFreq);

maxFreq-=1;

}

else stackMap.put(maxFreq, st);

}

}

return ans;

}

}