QN link : <https://www.codechef.com/problems/REMK>

Explanation of Qn :

* We need to find the maximum sum of subsequence of size k .
* The sum can be calculated as follows

G = [ (y0 + y1) + (y1 + y2) + (y2+ y3) + …….. + (yk-1 + yk) ]

Observation 1 : The sum g would be = y0 + 2 \* ( y1 + y2 + y3 + ….+ y k-1) + yk.

Observation 2 : We cant use sorting or priority Queue here , because it will find only top k elements .But we need subsequence of top k elements.

Observation 3 : Only the first and last element are fixed in the subsequnce , because this may possibly change the sum .

Example : [2 , 4 , 9 , 4 , 3 , 11 , 4 , 6] k = 4

Suppose if im taking 9 and 6 as first and last element , then 11 & 4 are the remaining element

Suppose if im taking 9 and 4 (6 th index) as first and last element .. till 11 & 4 are the remaining elements.

So the fist and last element is matter , the max elements are always same within those range.

**class Solution {**

**//Home work part**

**private int sumOfTopKElements(int firstIndex , int lastIndex , int k){**

**//Init a Max prirority Queue**

**Queue<Integer> pq = new PriorityQueue<>((a , b)-> Integer.compare(b , a));**

**for(int i = firstIndex ; i <= lastIndex ; i++){**

**pq.offer(nums[i]);**

**}**

**int sum = 0;**

**while(k-- != 0){**

**sum += pq.poll();**

**}**

**return sum;**

**}**

**public int maxSubsequence(int [] nums , int k){**

**int n = nums.length;**

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

**for(int j = i + 1 ; j< n ; j++){**

**int first = nums[i];**

**int last = nums[j];**

**// As we ve got the 2 elements , we only need get n - 2 elments**

**int k\_sum = topKelements(i + 1  , j - 1 , k - 2);**

**max = Math.max(max , k\_sum + first + last);**

**}**

**}**

**}**

**}**

There can be two soln for this one is using prirority Queue and another one is by sorting array

Time Complexcity : O ( N \* N \* N \* log k ) 🡪 O (N ^ 3 log k ) for sort [ O (N ^ 3 log N ) ]

Optimized Approch :

Observation :

* Here , we have the constrain that , a[i] <= 50 , so for each time we re selecting first and last element by ietaring over the array which takes the TC of O (n ^2)
* We can iterate over the values , to find the first and last element. So that we can reduce the TC by O(2500 \* n log k)

We know that the values are between 1 – 50 , but we dono whether it is exist in the array .For that we can use the HASHMAP .

Example : [ 8 , 8 , 7 , 9 , 9 , 12 , 12 , 9] k = 4

We cant choose 12 , 12 .Because there is no elements btw the sequence.

We can choose the pair 8 , 12 remaining element [9 , 12]

We can choose 8 , 9 remaining element [12 , 12]

But there are 2 eights and three nine’s .how we select which one is the best .

For maximize the sum , we need our search space should be larger to increase the possibility.So always select the first occurrence for the first element and last occurence for the last element.

class Solution {

    private int sumOfTopKElements(int firstIndex , int lastIndex , int k){

        //Init a Max prirority Queue

        Queue<Integer> pq = new PriorityQueue<>((a , b)-> Integer.compare(b , a));

        for(int i = firstIndex ; i <= lastIndex ; i++){

            pq.offer(nums[i]);

        }

        int sum = 0;

        while(k-- != 0){

            sum += pq.poll();

        }

        return sum;

    }

    public int maxSubsequence(int [] nums , int k){

        int n = nums.length;

        Map<Integer , Integer> firstOccurence= new HashMap<>();

        Map<Integer , Integer> lastOccurence= new HashMap<>();

        //Storing first and last occurence in a hash Map

        for(int i = 0 ; i < n ; i++){

            if(!firstOccurence.contains(nums[i])){

                firstOccurence.put(nums[i] , i);

            }

            lastOccurence.put(nums[i] , i);

        }

        //Instaed of iteraing over the array indexes , iterate over the values

        for(int i =50 ; i >= 1 ; i--){

            for(int j = 50 ; j >= 1 ; j--){

                int first = firstOccurence.get(i)

                int last = lastOccurence.get(j);

                // As we ve got the 2 elements , we only need get n - 2 elments

                 int k\_sum = topKelements(i + 1  , j - 1 , k - 2);

                 max = Math.max(max , k\_sum + first + last);

            }

        }

return max ;

    } }

Time complexcity : O (2500 \* n log k)

Since we have reduced the TC so much , eventhough it is not optimized one .

We need to still optimize the right part , we need to reduce the Tc for picking up the top k elements.

What shall we do ??

Iterate from 50 to 1 and get the occurrence of the element , reduce that occurrence from k and when k becomes zero breaks the loop.

But , there is big problem we don’t know the index of the occurrence whther it is between the range or exceeds the range right ?

We can create a 2D prefix sum array to store the occurrence of each element at particular index.

Example : [ 2 , 3 , 4 , 2 , 2, 5, 6 ,2 , 4 ]

Prefix[2] = [1 ,1 , 1 , 2 , 3 ,3 , 3 , 4 , 4]

Prefix[3] = [0 , 1 , 1, 1, 1, 1, 1, 1, 1 , 1]

Prefix[4] = [0 , 0 ,1 , 1 ,1 , 1, 1, 1 , 2]

Like this way , store the occurrence of all the element and we can retrieve it in O(1) time complexcity.

For retrieving the occurrence of the we can find it by prefix[lastIndex][j] – prefix[firstIndex – 1] [j]

Suppose , j = 2 that is we need to find the occurrence of the 2 element btw the index 4 and 7

Prefix[7][2] – prefix[4-1][2] 🡪 4 – 2 = 2

There are 2 two’s present between 4th index and 7 th index .

So , now we reduced the TC for picking top k elements too.

That’s all. We did it .

Time Complexcity : (50 \* 50 \* 50)

class Solution {

     int [][] prefix = new int[51][n];

    //Home work part

    private int sumOfTopKElements(int firstIndex , int lastIndex , int k){

        int sum = 0;

       for(int j = 50 ; j >= 1 ; j--){

         int occurence = prefix[j][lastIndex] - prefix[j][firstindex - 1];

         if(k >= occurence){

            sum += occurence \* j;

            k -=occurennce;

         }else{

            sum += (k \* j);

            k =0;

         }

         if(k == 0) break;

       }

       return sum;

    }

    public int maxSubsequence(int [] nums , int k){

        int n = nums.length;

        Map<Integer , Integer> firstOccurence= new HashMap<>();

        Map<Integer , Integer> lastOccurence= new HashMap<>();

        //Storing first and last occurence in a hash Map

        for(int i = 0 ; i < n ; i++){

            if(!firstOccurence.contains(nums[i])){

                firstOccurence.put(nums[i] , i);

            }

            lastOccurence.put(nums[i] , i);

        }

        // Calculating the occurence of the each element at every index

        for(int i = 1 ; i <= 50 ; i++){

            int count = 0;

            for(int j = 0 ; j < n ; j++){

                if(nums[i] == i){

                    count++;

                }

                prefix[i][j] = count;

            }

        }

        //Instaed of iteraing over the array indexes , iterate over the values

        for(int i =50 ; i >= 1 ; i--){

            for(int j = 50 ; j >= 1 ; j--){

                int first = firstOccurence.get(i)

                int last = lastOccurence.get(j);

                // As we ve got the 2 elements , we only need get n - 2 elments

                 int k\_sum = topKelements(i + 1  , j - 1 , k - 2);

                 max = Math.max(max , k\_sum + first + last);

            }

        }

        return max;

    }

}