Problem A. Maximize Frequency

Let us suppose the initial array contains distinct integers. We will extend the solution to the general case later on.

Observe that the frequency of any element of the array, say a_i , can be increased by performing the given operation only on the elements that are smaller than a_i. This gives us a hint to sort the given array. Let V be the sorted array. Then for each element v_i of V we need to find the smallest possible index $j(\leq i)$ such that the elements $v_j, v_{j+1} \dots v_{i-1}$ can be made equal to v_i in at most k operations.

Let k_p be the required number of operations to make v_p equal to v_i.

Then ,

$$\sum_{p=j}^{i-1} k_p \leq k$$

$$\Leftarrow \sum_{p=j}^{i-1} (v_i - v_p) \leq k$$

$$\Leftarrow \sum_{p=j}^{i-1} v_i - \sum_{p=j}^{i-1} v_p \leq k$$

$$\Leftarrow (i-j) \times v_i - (\sum_{p=1}^{i-1} v_p - \sum_{p=1}^{j-1} v_p) \leq k$$

$$\Leftarrow (i-j) \times v_i - (presum[i-1] - presum[j-1]) \leq k$$

Where , $presum[t] = \sum_{j=1}^{t} v_t$. Note that if v_t satisfies the last equation then all the elements $v_t, v_{t+1}, \dots, v_i$ also satisfy it. Also, if v_t does not satisfy the last equation then all the elements $v_1, v_2, \dots, v_{t-1}$ also do not satisfy it ($\because$, $presum$ is an increasing array). Hence, for each element v_i we can use binary search on the elements $v_1, v_2, \dots, v_{i-1}$ to find smallest possible index j such that v_j satisfies the last equation. The new frequency of element v_i will be $i - j + 1$. We can use map for storing the new frequency of the elements. We can then iterate through the original array and use the map to print the answer.

For a general array, we need to do the above algorithm only for the right most element of the group of equal elements in the sorted array V.

Time Complexity : O(nlogn)

Problem B. Worthy Arrays

As mentioned in the question, we cannot transfer values from negative elements therefore we only focus on positive elements initially given to us. Observe that if two elements are positive and we transfer some positive value from the first element to the second element there will be no difference in the absolute sum, that is, if $x = a[i]$ and $y = a[j]$ and $x > 0$ and $y > 0$ and $(i - j) <= k$ and any $0 < z < x$ then $|x| + |y| = x + y = |x - z| + |y + z| = x + y$.

Therefore we would try to transfer values from positive elements to some negative elements in the array so that the absolute sum could be minimised.

By doing the array traversal, let us say, we are at position i and the next nearest negative element is at position j such that $a[i] > 0$, $a[j] < 0$ and $abs(i - j) < k$. At such a point we would increment the $a[j]$ and decrement $a[i]$ by $min(a[i], abs(a[j]))$ which would be the most favourable case and similarly continue further.

Hence the time complexity: $O(n)$

Problem C. Longest Good Subsequence

We can solve it using dp. Define dp_i to be the length of longest good subsequence ending at value i. Since $0 \leq a_i \leq 1000$, our dp has only 1001 states. Initially $dp_i = 0$ for all i. Iterate through the array and for each a_i and do the change:

dp_{a_i}=$max\{dp_w + 1 \mid 0 \leq w \leq 1000; w \ \& \ a_i = w\}$.

While making the above change, store the value of w which maximizes dp_i to restore the subsequence later. Time Complexity: $O(n * MAX)$ where MAX is the maximum element of the array.

Problem D. Crack it

This problem can be solved using Dynamic Programming.

Let dp_i denotes the number of hackable keys having length i for $i > 0$ and $dp_0 = 1$. Let key_i denotes i^{th} character of key.

1.$i < k$

Any hackable key of length $i < k$ consists of only digits 0 and 1 since we can't have a group of letters having size k. So we have 2 options for each key_j $(1 \leq j \leq i)$.

$dp_i = dp_{i-1} \times 2$

2.$k \leq i$

Now hackable keys can have both digits and letters.

• If key_i is a letter $(a - z)$, key_j has to be a letter for $i - k + 1 \leq j \leq i$.

• Otherwise, key_i is either 0 or 1.

$dp_i = dp_{i-1} \times 2 + dp_{i-k} \times 26^k$

We can maintain a prefix array $preSum$ where $preSum[i] = \sum_{j=1}^{i} dp_j$ and $preSum[0] = 0$.

The answer is equal to $(preSum[l_2] - preSum[l_1 - 1])modulo 10^9 + 7$.

We will precompute dp_i and $preSum_i$ for $1 \leq i \leq N$ $(N = 10^5)$ so that each test case can be answered in $O(1)$ time.

Time Complexity: $O(N + t + logk)$

Problem E. Choosy Racers

Sort the cars by their speed, S_i in increasing order.

Now we greedily try to assign the cars to a driver in this increasing order.

When assigning the i-th car, consider all the unassigned drivers j with $A_j \leq S_i \leq B_j$. We claim that its optimal to pick the driver with the minimum B_j.

Suppose that both the drivers j and k can be assigned to car i, that is, $A_j \leq S_i \leq B_j$, $A_k \leq S_i \leq B_k$, and neither have been assigned to any car already. Assume $B_j \leq B_k$.

Suppose that k is assigned to i in the optimal solution, we show that we can get another optimal solution by assigning j to i without changing any of the previous assignments. Consider two cases,

1. If j is not assigned to any car. In this case, we can simply assign j to i and achieve another optimal solution.

2. If j is assigned to some car l. In this case, we can assign j to i, and k to l. This possible since $A_k \leq S_l \leq B_k$. We assigned the cars in increasing order of their speed, so $S_i \leq S_l \implies A_k \leq S_i \leq S_l$ and since j was assigned to l, $S_l \leq B_j \leq B_k$.

Thus, we can repeatedly swap the pairings and reduce the optimal solution to the greedy case.

We sort the drivers by A_j and maintain a multiset/priority queue of B_j's. When processing the i-th car, we add all B_j to the multiset such that $A_j \leq S_i$. If the set is empty or the minimum element is $< S_i$ we can't assign any driver to this car, otherwise we remove the minimum element and add it to the answer.

Note that each driver is added to the multiset atmost once.

Total time complexity is $O(C \log C + N \log N)$.

Problem F. How many friends?

We know that 2 children can be friends if their popularity value is a perfect square.

Let the prime factorization $a_x = p_1^a \ .p_2^b. \ ...p_k^c$, and similarly for a_y

Lets define mask(x)=$p_1^A \ .p_2^B .....p_k^C$ and similarly for a_y be mask(y) , where $A = a \ mod 2$,$B = b \ mod 2$....$C = c \ mod 2$.

For product of x and y to be perfect square we require mask(x)=mask(y); So we can reduce a_i = mask(a_i) and all those indices which have the same value can become friends with each other.

So basically the question reduces to finding the largest frequency of any element occurring in the updated array which can be done using a map or a count array/vector.

For speeding up the factorization part you need to use sieve and preprocess it upto N=10^6 which can be done in $NloglogN$ time. The overall Time complexity is O($NloglogN$ + $nlogn$) where N = 10^6.