Problem A (K^{th} Permutation Revisited)

Setter: Mohammad Ashraful Islam
Tester: Muhiminul Islam Osim,
Alter: Muhammad Ridowan, S.M. Shaheen Sha
Category: Ad Hoc

Editorial: You have find the summation of last M integers (numbers) of K^{th} permutation of consists of N integers numbered from 1, 2, 3, … , N. But, K $\leq$10 .
As, K $\leq$10, for any sequence, 1, 2, 3, 4, 5, …, N-4, N-3, N-2, N-1, N all the permutation/changes will be observed in last 4 integers (at most).
So, you can always take last $min(4, N)$ numbers and perform next permutation until you reach K^{th} permutation. Rest of the part can be done using a simple math formula to calculate the summation of series.

Problem B (Yet Another Suffix Array Problem)

Setter: Mohammad Solaiman
Tester: Raihat Zaman Neloy, Muhiminul Islam Osim
Alter: Md Mahamudur Rahaman Sajib
Category: Binary Search

Editorial: When we sort the subarrays lexicographically first the subarrays that start with 1 will come, then 2, then 3 and so on. When two subarrays have the same left endpoint, the one with smaller length is also lexicographically smaller than the other. For each value we know the number of subarrays that will have this value's position as the left endpoint. Calculate cumulative sum of these values. Now, we can do binary search on this array to find which value will be in the left endpoint of our desired subarray. After that, we can easily find the right endpoint too.

Problem C (Connect the Cities)

Setter: Raihat Zaman Neloy
Tester: Md. Imran Bin Azad, Muhiminul Islam Osim
Alter: Aminul Haq
Category: Greedy Observation

Editorial: If we observe the structure of the tree that satisfies the two conditions, we will find it to be a star. So, we just need to find a node which has N-1 nodes adjacent to it, now just take the sum of N-1 edges and find the minimum cost.

Problem D (Nim Heaps)

Setter: Pritom Kundu
Tester: Anik Sarker
Alter: Rafid Bin Mostofa
Category: Math

First let us solve the problem ignoring the condition that the array must be non decreasing. Suppose we have fixed the first i elements, let us count the number of ways we can choose the (i+1)th element. There are 2^i subsets consisting of the first elements. No two of them can have the same xor sum. The (i+1)th element cannot be one of these 2^i values. Thus there are $2^k - 2^i$ possible values for the (i+1)th element.

Now note that, no two elements can be equal (otherwise, those two have xor sum of 0). Thus every correct sequence will be strictly increasing and hence distinct. Thus we can simply divide our previous answer by $n!$. Thus our answer is

$$f(n, k) = (2^n - 1)(2^n - 2) \cdots (2^n - 2^{k-1})$$

This is an O(k) solution. To optimise it, let us rewrite it as follows.

$$f(n, k) = 2^{k(k-1)/2}(2^n - 1)(2^{n-1} - 1) \cdots (2^{n-k+1} - 1)$$
$$= F(k) \frac{G(n)}{G(n-k)}$$

Where, $F(k) = 2^{k(k-1)/2}$ and $G(k) = \prod_{i=1}^{k}(2^i - 1)$

Both F and G can be precalculated in O(k) time beforehand. Then queries can be answered in O(1).

Problem E (Min Cost Sort)

Setter: Arghya Pal
Tester: Nafis Sadique
Alter: Hasnain Heickal, Muhiminul Islam Osim
Category: Greedy, Graph, Math

The key to solve this problem is to think in terms of the permutation cycle. For the cycle containing 1, it's always better to swap the element with 1. For other cycles, either merge it with 1's cycle initially or make all the swaps with the smallest element in the cycle. Do which one gives you the smaller cost.
https://ideone.com/XRBZim

Proof: We'll think about everything in terms of the permutation cycle. **Any swap either breaks one permutation cycle into two or merges two permutation cycles into one.**

So we are given some cycles, our goal is to do some operations to reduce each permutation cycle to size 1.

Let's think what is the optimal way to break a cycle of length k into k cycles of length 1(ignore everything outside this cycle for now). Assume the elements are a_1, a_2….a_k. We need at least k-1 operations. The ultimate sum would be result of at least 2*(k-1) numbers and each of a_1, a_2…..a_k would exist at least once in the sum. And the remaining k-2 numbers would be no smaller than min(a_1, a_2…..a_k). So the sum can not be smaller than

$$a_1+a_2+....a_k + (k-2)*\min(a_1, a_2.....a_k)$$

and this is achievable. This is important and the whole problem is kind of dependent on this idea.

Assume there are k cycle's initially.
s_i = set of all elements initially in the i'th initial cycle
m_i = min of all elements in s_i
$|s_i|$ = size of set s_i
sum(s_i) = sum of all elements of in set s_i

We use set s_i to denote the i'th initial cycle to avoid confusion as cycles are merging and breaking.

Consider an optimal sequence of sorting. In each operation we are breaking a cycle or merging two cycles. Consider a graph G with k nodes where each node represents a set and an edge between node u and v represents that after some operation during the sort process, some element from s_u and some element from s_v got into the same cycle.

This graph has some connected components, different connected components don't interfere with each other, so we can think of them independently.
Let's say there is a connected component with c nodes and we try to calculate the lowest cost for sorting elements within this component's set. Assume the sets associated with nodes in this component are s_1, s_2…s_c.

As there are c nodes, then there must have been at least c merge operations and as at least one node from each set has participated in the merge operation,
minimum cost for merge operations would be >= $m_1+m_2+...m_c + (c-2)*\min(m_1,m_2,...m_c)$

There would be at least $|s_1|+...+|s_c|$ - 1 break operation and again all the elements in all the sets must participate in the break operation at least once, so minimum cost for breaking would be >= $sum(s_1)+...+sum(s_c) + (|s_1|+...+|s_c| - 1)*min(m_1,...m_c)$.

And both of these minimums are achievable.

Let's call M = $min(m_1,...m_c)$ and S is the set containing M.

What we can do is that we merge all the other sets into C. Then just start breaking this big cycle, every operation would be between M and some other element x, resulting in separating x from the cycle.

So the big picture turns out to be that we can achieve the lowest cost by merging all the sets in this specific connected component(except S) into the set containing the smallest element(which is S here). And then separating out each element one by one. So from the standpoint of a specific set s_i(except S), we can view it like this$_i$ at first we are merging it into S with cost $M+m_i$ and then making $|s_i|$ operations with total cost $|s_i|*M + sum(s_i)$, occuring a total cost of,

$$M*(|s_i|+1) + sum(s_i) + m_i$$

And for S we are getting cost

$$sum(S) + (|S|-2)*M.$$

This point of view enables us to get rid of mathematical equations from the rest of the proof.

So what we have established till now is that, there exists an optimal sequence of operation where we can group the permutation cycles initially, then for each group we'll just merge all cycles into the cycle containing the smallest number and then break that big cycle one element at a time. And different groups don't interfere with each other. Also cost for these operations can be thought as independent per cycle. Now it is easy to see that for any cycle it is better to merge it to 1's cycle than any other cycle. So for each group we can just break the cycle containing the smallest number within itself and merge the rest of them to 1's cycle and break it later. So how to check weather for a cycle s_i if we should merge it to 1's cycle or break it within itself?

Breaking it within itself gives us cost $|s_i-2|*m_i + sum(s_i)$

And merging it into 1's cycle and later breaking it gives us cost, $1+m_i + |s_i| + sum(s_i)$

Just do whichever gives you the smaller cost.

Problem F (Power Sequence)

Setter: Md Mahamudur Rahaman Sajib

Tester: Anik Sarker, Pritom Kundu

Alter: Rafid Bin Mostafa
Tags:Dynamic Programming, Math, Online FFT

here beatiness *F* function can be written as,

$F(\{x1,\ x2,\ x3 \ldots . xk\}) = B[x1] * (1 + B[x2] * (1 + B[x3] * \ldots..))]$

Let's try to solve this problem for m = 1.

$dp[n] = sum\ of\ beautiness\ function\ F\ for\ all\ possible\ such\ that\ sum\ is\ n$

$$dp[n] = \sum_{i=1}^{n} B[i] * (1 + dp[n - i])$$

Here time complexity of this naive dp solution is $O(n^2)$.

But if we think of 3 polynomials,

$$G(x) = \sum_{i=0}^{n} B[i] * x^i$$

$$H(x) = \sum_{i=0}^{n} (1 + dp[i]) * x^i$$

$$Q(x) = \sum_{i=0}^{n} dp[i] * x^i$$

then we can easily see,

$Q(x) = G(x) * H(x)$

But the problem is G(x) is static but H(x) is dynamic and H(x) itself depends on the Q(x) so polynomial multiplication using FFT is not possible here. To solve this problem faster way we need to use the trick of online FFT(where we can trickily do polynomial multiplication block by block where block size will be power of 2)(Online FFT), then time complexity will be reduced to $O(nlog(n)^2)$.

But, how can we solve the problem for any m? First, the observation is that m is not too big. Let's try to sketch the dp formula.

$dp[m][n] = sum(F(seq)^m)\ for\ all\ possible\ sequence\ such\ that\ sum\ is\ n$

$$dp[m][n] = \sum_{i=1}^{n} B[i]^m * \sum_{j=0}^{m} mCj * dp[j][n - i]$$

Time complexity of this dp solution is $O(n^2 * m^2)$ which is too slow.

Let's rewrite this equation for m = p,

$$dp[p][n] = \sum_{i=1}^{n} B[i]^p * \sum_{j=0}^{p} pCj * dp[j][n - i]$$

For different p, we can derive polynomials

$$G_p(x) = \sum_{i=0}^{n} B[i]^p * x^i$$

$$H_p(x) = \sum_{i=0}^{n} (\sum_{j=0}^{p} pCj * dp[j][i]) * x^i$$

$Q_p(x) = G_p(x) * H_p(x)$

so, here we need to do online fft for different p which is a bit tricky to code. But there is a small problem to build $H_p(x)$. If we iterate over j every time we build $H_p(x)$ so time complexity of total builds of $H_p(x)$ is $O(nm^2 log(n))$ and online FFT part's time complexity will be $O(nmlog(n)^2)$. So time complexity is $O(nm^2 log(n) + nmlog(n)^2)$.

Problem G (Creative Kindergarten)

Setter: Mohammad Ashraful Islam
Tester: S.M. Shaheen Sha
Alter: Mohammad Ridowan
Category: Adhoc

Problem H (Is this Graph?)

Setter: Aminul haq
Tester: Arghya Pal
Alter: S.M. Shaheen Sha, Md. Mahamudur Rahaman Sajib, Raihat Zaman Neloy
Tags: Data Structure, DSU+Link Cut Tree or DSU+HLD+Segment Tree/BIT
Solution: There are several solutions to solve this problem, we will discuss the offline approach using DSU and HLD.
First, let's understand the problem. It simply asks to count the number of bridges between two nodes in each query. But the challenge is the updates. When we are adding a new edge between two nodes, and if those two nodes are already connected, then after adding it, there won't be any bridges between them, as it will become a biconnected component.
So, to solve it, first, we will read all the full input and make the tree (or forest). How to do it? Using DSU, we will add an edge to our tree if those two nodes are not connected already. Now we have a tree (or several trees which make a forest). We can build an HLD structure from these trees. Now we will iterate over the entire input again. When we get a tree-edge (from previous DSU) we will set this path's value to one. And when we get a non-tree-edge, we need to set zero to all the edges in that path, as this new edge will remove all the bridges between these current two nodes. And for a query will count how many 1's are there in that path (or simply, the total sum of values in that path).

https://ideone.com/6CbfDh
https://ideone.com/3ACBYU

Problem I (Beautiful Blocks (Easy))

Setter: Ashiqul Islam
Tester: Arghya Pal
Alter: Pritom Kundu
Tags: Observation, DP
Solution:

Let's say,
A block is X-block if the path has X cells in the block.

The path goes through exactly (N-1) blocks.
Among them:

(1) K blocks are 3-block
(2) (K-1) blocks are 1-block
(3) the rest are 2-block

So, we can dp on,
(block_index, how many blocks has been taken, how many 3-blocks have been taken, how many 1-block has been taken)

the answer would be f(no_of_blocks, N-1, K, K-1) for all K.

this is too slow. instead of keeping both the count of 3-blocks and 1-blocks, we can keep only their difference.

(block_index, how many blocks has been taken, difference between 3-block count and 1-block count)

This is a n^4 dp. This is also too slow.

Instead of considering all the blocks in the dp, we can only consider O(N) blocks. The relevant blocks can be found as follows:

Let's say for block, a1 = its max, a2 = 2nd max, a3 = 3rd max, a4 = 4th max.

Let's sort the blocks three times in decreasing order of: a1, (a1+a2), (a1+a2+a3).

The relevant blocks are the union of the first (n-1) blocks from all the 3 sorted arrays.

Overall Complexity (n^3)

Interestingly, there is another solution. You can notice that in an optimal selection, the number of 3-blocks and 1-blocks are almost equal. So if you random shuffle all the blocks, then the max difference of 3-blocks and 1-blocks in an optimal assignment over all the prefix won't be very big. So you can reduce that difference dimension to something like 20.

Github Link:
https://github.com/Contest-Problems/beautiful_blocks

Problem J (Beautiful Blocks (Hard))

Setter: Arghya Pal & Ashiqul Islam
Tester: Arghya Pal
Alter: Pritom Kundu
Tags: Observation, DP
Solution:

Now the solution for easy version is too slow for the harder version. Let's solve for K = 1. You can easily make it work for the given Ks.

Let's sort the blocks in descending order of sum of their best+2nd best block. Now we can observe that for any two indices i,j such that (1<= i < j <= n), it's always better to take the i'th block as 2-block rather than the j'th block. So there exists a prefix of the blocks where we will take all the blocks and from the rest we will only take 1-blocks and 3-blocks.

So we need to calculate two things for each prefix,
i) if we take all the blocks in this prefix, for each different value of (3_blocks - 1_blocks), what is the maximum we can achieve? We can do it in O(n^2) complexity.

ii) what is the maximum we can achieve if we take a fixed number(N-1 - length of the prefix) of blocks from the rest of the blocks and again we need to know it for all different values of (3_blocks - 1_blocks). It turns out that for a suffix we can calculate this value in O(nlogn) complexity. So for n suffixes, total complexity sums up to be O(n^2*logn).

Let's see how we can do the 2nd part.
Now assume we are given **m** blocks, we have to take exactly k block from them and we need to know, what is the maximum we can achieve from

(1) k 1-blocks and 0 3-blocks
(2) k-1 1-blocks and 1 3-blocks
.....
.....
.....
(k+1) 0 1-blocks and k 3-blocks

If we have to take k 1-blocks and 0 3-blocks, it's obvious we will take the k blocks which have the highest a1(a1 denotes the maximum cell in the block).

Now if we need to take k-1 1-blocks and 1 3-blocks, we can either turn one 1-block into 3-block or remove a 1-block and take a previously skipped block as 3-block.

In general if we have a set of 1-blocks and 3-blocks which gives us best result for (x,k-x) configuration(x 3-blocks and k-x 1-blocks), then for obtaining best result for (x+1,k-x-1) configuration we can just turn one 1-block into 3-block or remove a 1-block and take a previously skipped block as 3-block. This idea seems very intuitive but the life of a problem-setter is not so easy, you have to prove it also :(

(Un)fortunately the proof is ~~easy~~ hard in this case.(changed my mind while writing the proof, not easy to write, definitely not easy to read)

Essentially, what we are doing is that while going from (x,k-x) configuration to (x+1,k-x-1) configuration, we are keeping the set of 3-blocks intact and just adding another 3-block, also just changing exactly one 1-block(either turning it into 3-block or removing it).

Lets assume,

A is the set of skipped blocks, B is the set of 1-blocks, C is the set of 3-blocks for some (x, k-x) configuration. .....(assumption 1)

One thing that we can observe is that going from (x,k-x) to (x+1,k-x-1) can be done without any “direct exchange” between sets.

Direct exchange between two sets X, Y takes place if at least one block from X goes to Y and one block from Y goes to X.

Let’s say A_1, B_1 and C_1 corresponding sets for 0-blocks, 1-blocks and 3-blocks which gives us maximum result for (x+1, k-x-1) configuration and among all such sets C_1 is a set such that $|C_1$ intersection $C|$ is maximum. If C is not a subset of C_1, that means there was at least one block c in C that is now at some other set.

i) If c is now in B_1, then
   1) Either there exist a block b in B which is now at C_1, or
   2) There exist a block b in B which is now at A_1 and there is a block a in A which is now at C_1

ii) If c is now in A_1, then
   1) Either there exist a block a in A which is now at C_1, or
   2) There exists a block a in A which is now at B_1 and there is a block b in B which is now at C_1.

In all these cases we can cyclically reverse the position of the blocks and we will achieve a better or equal result.(Otherwise assumption 1 can not be true)

We can do this as long as C is not a subset of C_1, every time we will be increasing the $|C$ intersection $C_1|$.

In the same way, we can see that if any block goes from B to A_1 and a block goes from A to B_1, we can reverse this as well.

Github Link:
https://github.com/Contest-Problems/beautiful_blocks