Problem Tutorial: "Bananas Packing"

The greedy solution is a correct approach: while there is a way to pack a box of seven bananas, take the one with the smallest d. To do that iterate over all d in increasing order, and calculate how many boxes with the exact d value — there are $\min(a_d, a_{d+1}, \ldots, a_{d+6})$ such boxes. Just add these to the answer, and subtract the value from $a_d, a_{d+1}, \ldots, a_{d+6}$. The number of remaining bananas is the sum of all a_i in the end.

Problem Tutorial: "Permutations"

Let's process every column one by one. If a column has a repeated number in it, it can not become a permutation, otherwise, let's find the unique missing number. We can implement it using an auxiliary array $col\_cnt[i]$ of length n, storing the number of times i appeared in the column.

Notice that if multiple columns are missing the same number, then we can make at most one of those columns a permutation. After choosing one of these columns, we can put any numbers into other columns. This means that if there are k columns missing the same number x, then all permutations that have x in one of those k places make one column a permutation.

Let's create an array $cnt[i]$ equal to the number of columns missing number i.

- If $cnt[x] == 0$, then it does not matter where the x is placed in the last row. Let's calculate the number of zeroes in cnt, call it $zeroes$.
- If $cnt[x] \neq 0$, then we have $cnt[x]$ places to put x in the last row to get an additional column-permutation. Let's calculate the product of all non-zero values of $cnt[x]$, call it $ways$.

Then the maximum total number of possible column-permuitations is $n - zeroes$, and the number of permutations that achieve this answer is $ways \cdot zeroes!$.

Problem Tutorial: "Equation"

Let's focus on a case $l = r$, this means that $b + c = l$. Recall Vieta's formula $b = -x_1 - x_2$, a $c = x_1 \cdot x_2$.

Rearrange the terms in this expression:

$$x_1 \cdot x_2 - x_1 - x_2 = l$$

$$x_1 \cdot (x_2 - 1) - x_2 = l + 1 - 1$$

$$x_1 \cdot (x_2 - 1) - x_2 + 1 = l + 1$$

$$x_1 \cdot (x_2 - 1) - (x_2 - 1) = l + 1$$

$$(x_1 - 1) \cdot (x_2 - 1) = l + 1$$

Turns out, $x_1 - 1$ and $x_2 - 1$ must divide $l + 1$. The number of pairs of roots is infinite when $l = -1$, because if we set $x_1 = 1$ the equation becomes $1 \cdot x_2 - 1 - x_2 = -1$, which is true for all $x_2 \in \mathbb{Z}$. For every other l the answer is finite.

Now our problem is to find the number of ways to represent $l+1 = x_1 \cdot x_2$ as a product of two integers. We can take x_1 to be any integer divisor of $l+1$ (including negative divisors), but different ways to choose x_1 might result in the same pair. All pairs of divisors of $l+1$ can be divided into pairs (x_1, x_2) and (x_2, x_1),

except when these two pairs are the same. This is true when $x_1 = x_2$, meaning that $x_1^2 = l + 1$, that is, $l + 1$ is a perfect square. Finally, the answer for l is the number of divisors of $l + 1$, plus one in case l is a perfect square.

To calculate the answer for $[l, r]$ we will use modified Eratosthenes' sieve. The number of divisors is equal to the product of prime exponents plus one. First, we'll find all primes up to 10^6. Iterate over these primes, then we'll process all numbers inside $[l, r]$ that are divisible by $prime_i$. To process the numbers, loop over all numbers on $[\lceil \frac{l}{prime_i} \rceil \cdot prime_i, \lfloor \frac{r}{prime_i} \rfloor \cdot prime_i]$ with step $prime_i$ and divide the current numbers while it is divisible. Multiply the number of divisors for this number by the number of times we divided it by $prime_i$ plus one.

The first and the last numbers on $[l, r]$ that are divisible by $prime_i$ are $\lceil \frac{l}{prime_i} \rceil \cdot prime_i$ and $\lfloor \frac{r}{prime_i} \rfloor \cdot prime_i$ respectively. After we have processed all primes up to 10^6, we still have to process all the primes larger than 10^6. Each number on the segment has at most one such prime left in its factorization.

Problem Tutorial: "Fantastic Three"

Look at some fixed value of j. Let's try all possible lengths of a common prefix that $(a_i \oplus a_j)$ and $(a_j \oplus a_k)$ can have. Notice that in this case we can "cancel" a_j, so a_i and a_k have the same common prefix. The bit value right after the common prefix should be 0 in $(a_i \oplus a_j)$ and 1 in $(a_j \oplus a_k)$. This means that if a_j has 0 in that place, then a_i has to have 0, and a_k has to have 1. Similarly, if a_j has 1 in that place, then a_i has to have 1, and a_k has to have 0.

Let's iterate over all j in increasing order and maintain two multisets of values: $P = \{a_1, a_2, \ldots, a_{j-1}\}$, $S = \{a_{j+1}, a_{j+2}, \ldots, a_n\}$. We also need to be able to perform the following operations:

1. Add or remove an element from P.
2. Add or remove an element from S.
3. Calculate the number of pairs $p \in P$, $s \in S$, such that p and s are equal until the k-th bit, and p is smaller/greater than s in the k-th bit.

We will maintain the number of pairs for the third operation in a separate array. To be able to do it, we need to update this array during the first two operations. Let's build a bit trie for elements from both P and S, most significant bits first.

If we imagine that we already have a trie, then with one trie traversal for every vertex v we can calculate the numbers of elements from P that end in a subtree of v (similarly for S). Then, to calculate the array of answers, we need to sum the product of the number of elements of S located in the left subtree of v times the number of elements of P located in the right subtree of v (or the other way around). Whenever we add/remove an element to/from this trie, these products only change for vertices we visit during the addition/removal of that element. Only recalculate products for these vertices during the update. This enables us to execute an addition/removal query in $O(\log C)$ time, where C is the upper bound on a_i. The overall time complexity of this solution is $O(n \cdot \log C)$.

Problem Tutorial: "Reconstructing Pairs"

This problem was about "directing" the given n input pairs such that the first elements of these pairs make up the multiset A and the second elements make up the multiset B. To solve the first subtask you just need to try all possible orientations in $O(2^n)$ time.

Let's state this problem as a graph problem. Build a graph with all possible values in the input as vertices, and with given n input pairs as edges (loops and multiple edges are possible). Initially undirected edges must be directed in a way that every incoming degree is equal to the number of occurrences of this number in A, and outgoing degree is equal to the number of occurrences in B. In particular, the (undirected) degree

of every vertex must be equal to the total number of occurrences of that number in A and B combined. Check this condition before moving on.

Notice that if this check passed (total degree is correct for every vertex), and the orientation of edges satisfies the incoming degree constraints (which correspond to multiset A), then outgoing degree constraints are automatically satisfied. From now on, we only focus on incoming degree constraints. Now all that's left is to choose one of the ends for each edge such that every vertex is chosen the correct number of times.

This problem can be solved by finding the perfect matching in a bipartite graph: vertices in left part correspond to elements of A (with respect to multiplicity), vertices in the right part correspond to input pairs and are adjacent to two vertices in the left part (which are the elements of this input pair). This bipartite graph has a perfect matching iff our problem has a solution. The graph has $O(n)$ vertices and $O(n^2)$ edges, but Kuhn's algorithm is fast enough to solve the second-to-last subtask.

If we merge the vertices corresponding to the same number, we get a graph with $O(n)$ edges, but now each vertex in the left part has a multiplicity: how many vertices in the right part it should be matched to. To solve this problem you can use a modified Kuhn's algorithm that starts the DFS from the same vertex multiple times. If optimized, this can be fast enough to score 100 points. The other approach is to use Dinic's algorithm for solving the maximum flow problem. Our graph satisfies the Karzanov's theorem, so the time complexity of Dinic's algorithm is $O(n\sqrt{n})$.