Problem 1A. Air Flights

The required answer is $52 \cdot \left(\sum_{i=1}^{7} a_i \right) + a_5$. This formula comes from idea that we have 52 weeks in a year, and every week there are $\sum_i a_i$ flights in total. If we agree to start week on friday, we can go through 52 weeks ($52 \cdot 7 = 364$), and one extra day (365th) is friday. So we add friday's number of flights to answer one more time.

Problem 1B. Blooming Sakura

Note that minimal sum a_v for all vertexes from 1 to n equals to $n-1$ (when parent of vertex 2, vertex 3, ..., vertex n is vertex 1), and maximal sum equals to $\frac{n(n-1)}{2}$ when our graph is a "bamboo"with n vertexes, where parent of vertex 2 is vertex 1, parent of vertex 3 is vertex 2 and etc. Then, if $k \leq n-2$ or $\frac{n(n-1)}{2} + 1 \leq k$ the answer is NO. Otherwise, the answer is YES. Now we can make a correct tree following next algorithm: Let us assume that parent of all vertexes is vertex 1. Then current *sum* equals to $n-1$. Lets look through all vertexes from 2 to n and try to hang them up to new vertexes until we get the second point of algorithm. Let we want to hang uo the vertex v, so lets look can we add $v-2$ to the current sum.

- If we can, then hang up vertex v to vertex $v-1$, so we add $v-2$ to current sum because parent of $v-1$ is $v-2$, parent of $v-2$ is $v-3$ and etc. It means that on a way from $v-1$ to 1 we have $v-1$ vertexes. Continue algoritrhm with vertex $v+1$.
- If we can't it means that $k \leq sum + v - 3$, so we can hang up vertex v to vertex $k - sum + 1$, and others vertexes $v+1$, $v+2$,..., n keep suspended to vertex 1. In this case, sum increases to k. Because our algorithm didn't finish on vertex $v-1$, so we can hang up in that way. Lets finish our algorithm.

Problem 1C. n-dimensional chocolate

Numbers which are not greater than 10^{16} have at most 41472 divisors. Furthermore sum of number of divisors of divisors doesn't exceed 36374184. Denote D as number of divisors of k and the previous sum as CNT.

The chocolate has no more than $log_2(10^{18}) \approx 60$ sides with size greater than 1. Let's enumerate all divisors (array D) and use dynamic programming. To do that we can factorize k in $O(\sqrt{k})$.

Denote $dp[i \leq n][j \leq |D|]$ as maximum volume of minimal piece in the first i dimensions after D_j cuts in the same dimensions. There are $log_2(C) * |D|^2$ transitions in this dynamics. This solution passes all groups except the last two.

To pass the last group let's find all pairs (i, j) that $D_i | D_j | k$. Denote $div[i][p]$ as set of indices j there D_j divides D_i and $\frac{D_i}{D_j}$ has no prime divisors greater than p. We are interested in primes that are divisors of n. To calculate these sets we can iterate in increasing order of p, go from left to right and add div[index of $\frac{D_i}{p}$][p] to $div[i][p]$. Let's notice that the set of all divisors D_i will be equal $div[I]$[maximum prime divisor]. Also, we can store only one layer that corresponding to last calculated prime number. All sets can be calculated in $O(|D| \cdot |P| + CNT)$. Moreover, we can merge sorted sets, get as a result a sorted set and algorithms still works in $O(|D| \cdot |P| + CNT)$.

Let's use calculated sets in the previous dynamics. There will be $log_2(C) * CNT$ transitions in this dynamics. This solution passes all groups except the last one.

To pass the last group we will use that the indices are sorted in increasing order of D_j so that if for some j it turns out that the size of the current side is $A \leq \frac{D_i}{D_j}$, then the subsequent j can be ignored. Then for each A we consider exactly $|D| + \#(D_i \cdot D_j | k$ and $D_j \leq A)$, and this stage of the algorithm will take $O((|D| + \#(D_i \cdot D_j | k$ and $D_j \leq A) \cdot \log_A 10^{18})$.

Let's try to estimate this number. It is easy to see that:

If $A \le 100$, then $(|D| + \#(D_i \cdot D_j | k \text{ and } D_j \le A)) \cdot \log_A 10^{18}) \le |D| \cdot 100 \cdot \log_2 10^{18}$.

otherwise $(|D| + \#(D_i \cdot D_j | k \text{ and } D_j \le A)) \cdot \log_A 10^{18}) \le CNT \cdot \log_{100} 10^{18}$ this estimate is inaccurate, but it is an upper estimate, and in our case it is enough.

Final asymptotics: $O(\sqrt{k} + |D| \cdot |P| + CNT + CNT \cdot \log_{100} 10^{18})$;

Problem 1D. Journey planning

First let's think of $O((n+K) \cdot n \cdot C)$ solution, where $a_i, b_i \le C$. It's simple dynamic programming — denote $dp_{i,j} (0 \le i \le n, 0 \le j \le n \cdot C)$ as maximal pleasure of your friend in first i days if your pleasure is exactly j. This solution scores 60 points.

Then you need to make some observations. There are some cases when the answer is trivial: if you choose the most pleasant attraction for yourself each day and your friend still gets more pleasure, then that is the optimal choice. Another simple case is symmetric: if you choose the most pleasant attraction for your friend each day and you still get more pleasure.

If the answer is neither of two cases, we always can decrease your pleasure and increase the pleasure of your friend, and symmetrically increase your pleasure and decrease the pleasure of your friend. If you increase someone's pleasure by changing choice at one day, it increases for at most C and others' pleasure decreases for at most C. So, in optimal answer difference between the two pleasures is at most $2 \cdot C$.

Now we can shuffle days in random order and use dp solution with extra limit : $0 \le j \le 4 \cdot C \cdot \sqrt{n}$. Since days are shuffled, the probability of error is negligibly small (to prove it, we can use the one-dimensional random walk concept).

So, we have an $O(n+k) \cdot 4 \cdot C \cdot \sqrt{n}$ solution, which is fast enough to get 100 points.

Problem 2A. Simple Pattern

There are two cases in the problem: pattern has no characters «*» or pattern has at least one. If pattern is fixed and we already know N, we should check if N is composite with $O(\sqrt{N})$ algorithm.

If pattern has exactly one character to replace, we can make our number dividable by 3: we just need sum of digits to be dividable by 3. So we need to calculate sum of digits and find remaining one solving equation $sum + digit \equiv 0 \pmod 3$. Of course, we need only one solution and 1, 2, or 3 will always be a solution for $digit$. This algorithm works in linear time from number of digits in N, which is $O(\log N)$.

If we had more «*» characters, than one, we can easily replace all but one of them with random digit and use previous solution with new pattern.

Problem 2B. Going Home

Let's assume that cities are vertices, and the paths between them are edges. Then we try to understand what conditions are necessary to create a sequence of actions that leave exactly 1 edge in the end.

First note: all vertices that have edges must always be connected. This is necessary because in one operation we add only an edge between the vertices of same component, so in the end we will have a number of edges at least equal to the number of components, but we need exactly 1 edge in the end.

Second note: the number of vertices that have an odd number of edges should be no more than 2. Let's understand why this is necessary: in one operation, the number of paths changes only at the intermediate vertex, the number of edges from this vertex decreases by exactly 2. Thus, we will have at least 1 edge from all vertices that initially had an odd number of edges, so if there are more than 2 such vertices, then we will end with more than 1 edge.

If any of the conditions above are false, then there is no answer. Now let's understand that if all conditions are true, then we can build an answer. To do this, we will find an Eulerian path in the graph (a path containing each edge exactly 1 time), because these conditions are also sufficient for construction this path. Then sequentially using the vertices of this path $v_1, v_2, ..., v_m$, we will go along i from 2 to $m-1$ and turn the edges (v_1, v_i) and (v_i, v_{i+1}) to edge (v_1, v_{i+1}).

Problem 2C. Strange Sum

Let's understand how to calculate sum from the statements in a bit different way. To do this, we can use dynamic programming and calculate $dp[i] = dp[i-1] + i \cdot (i+1)/2$, $dp[i]$ is the number of positions that will be compared for equality for an subarray of length i. Now we will calculate another sum where instead of Hamming distance we will check how many positions in two subarrays are equal. Then the original sum on a subarray of length len will be equal $dp[len]$ — the new sum. Now we are working only with the new sum. To find this sum, consider pairs of positions in which the characters are equal, so let's fix the indices i and j so that $a_i = a_j, i \le j$. Then this pair add to sum number equal $len - j$. To understand this, for simplicity, consider positions of array from 0 to len, then a pair of indexes i and j will add 1 only for subarrays $([0; i], [j-i; j]), ([0; i+1], [j-i; j+1]), \ldots, ([0; len-(j-i)], [j-i; len])$. So to solve the problem, it is enough to calculate the sum of $len - j$ for all pairs.

Now let's show how to calculate such a sum using the MO algorithm. Do not forget to compress the numbers, so we will get no more than n different digits and can use an array instead of a hash map to save some statistics for digits. Let's calculate $lastEq[i]$ - the nearest position before i, in which there is a digit a_i. All next statistics we calculate for subarray on the current operation of MO algorithm. We will support the following values: $lastNumber[i]$ is the index of the last element on the segment equal to i, $cnt[i]$ is the number of digits equal i on the segment, $lens[i]$ is the sum of the distances between each of the digits i to the last digit i, a variable $cntPair$ is the number of ordered pairs of positions in which the digits are equal and a variable ans is the current sum for subarray.

Then, when we move the right border to the right, we will update all statistics variables in an understandable way, $lastNumber[a_r] = r$, $lens[a_r]$ += $cnt[a_r] * (r - lastEq[r])$, $cnt[a_r]$ += 1, $cntPair$ += $cnt[a_r]$ and after that $cntPair$ will be added to the ans (for each pair we increase amount of available subarrays by 1). Similarly, when moving the right border to the left, we do same operations in reversed order and instead using += we use −=, also $lastNumber[a_r] = lastEq[a_r]$ if this value is greater than left border.

When moving left border to the left, the values of the variables will be updated in the next way, $lens[a_l]$ += $lasNumber[a_l] - l$, $cnt[a_l]$ += 1, $cntPair$ += $cnt[a_l]$ and after that $lens[a_l] + cnt[a_l] * (r - last[a_l])$ will be added to the answer (add sum for each pair with first position equal l). Similarly, when moving the left border to the right, we also reverse operations and use −= instead of +=.

Problem 2D. A third grade problem

The main observation is that n can be produced by $\log n$ multiplications and $2 \log n$ additions using binary representation. So $a_i \le 2 \log n$ or answer to a query less than $\log n$. Moreover it can be shown, that number of multiplications always less than $\log n$, but it was unnecessary to solve the problem. Let's solve a case there number multiplications is less than $\log n$. In other case solution will be mostly the same. Denote $dp[i][j]$ as minimum number of additions needed to generate j if we can use at most i multiplications. There are two types of transitions: the last operation is multiplication and the last operation is addition. It the first case we will go through all possible halfs which product is not greater than maximum value of n. So this part have asymptotics equal to $O(n \log^3 n)$. In the second case the main idea is one of terms always can be less than 5, so again we go through and result asymptotic equals to $O(n \log^2 n)$. Finally, each query can be answered in $O(\log n)$ by going thought all optimal pairs of (addition, multiplications).