Problem A. Trucks

This problem is solved using a greedy algorithm. Before applying it, the array of truck costs must be sorted in ascending order. Since the array is quite large, a sorting algorithm with a time complexity of $O(n \cdot \log n)$ should be used. Next, we need to find the maximum value of k such that $\sum_{i=0}^{k} a_i \leq M$. The number k is the answer of the problem.

Problem B. 2025

There are several different methods to solve this problem. Let's describe two alternative methods.

First method of solution

For $k = 1$. In this algorithm, we add 1 to the number N. After each addition, we check for the occurrence of digits 2, 0, 2, 5 in the corresponding order in the new number N. Unfortunately, already at $k = 2$ many tests will not pass within the time limit.

Let's call k repetitions of numbers 2025 the «sought series». In the second algorithm, we check the number of t digits from «sought series» of number N.

Consider, for example, the sequence 2025. Let the number of digits in number N be greater than four (when the number of digits is less than or equal to 4 — the solution is obvious). Suppose $t = 0$. In this case, we check the number formed by the last 4 digits of number N. If it is less than 2025, then we replace them with digits 2, 0, 2, 5. If the number formed by the last 4 digits of number N is greater than 2025, then we add 10000 to number N and replace the last 4 digits with 2, 0, 2, 5. Consider the case when $t > 0$. In this case, we consider the number composed of the last $4 - t$ digits of number N and compare it with the number composed of the last $4 - t$ digits of number 2025. If the first number is smaller, then we append the second number to it on the right. Otherwise, we find the number $N_1 = N + 10^{4-t}$ and consider the number of digits of sequence 2025 in the integer part of number $N_1/10^{q-t}$. Let's denote this number by p. We compare numbers t and p. If $p = t$, then the answer will be the number N_1, in which the last $q - t$ digits are replaced with the last $q - t$ digits of number 2025. If $p = t + 1$, then the answer will be the number N_1, in which the last $q - t$ digits of number N_1 are replaced respectively with '0' and the last $q - (t + 1)$ digits of number 2025. If $p = t + 2$, then the answer will be the number N_1, in which the last $q - t$ digits of number N_1 are replaced with '00' and the last $q - (t + 2)$ digits of number 2025. If $p = t - 1$, we apply recursion. Sequences of the form 20252025 and 202520252025 are considered similarly.

Second method of solution

First, write the number 2025 repeated k times in a row. If this number is already greater than the number N, then we've found the answer. Otherwise, if that's not the case, then the desired number M must either have the same length as N, or be one digit longer.

Let's consider the case where the length of the desired number M is equal to the length of N. Clearly, some prefix of M must match a prefix of N. Let's iterate over that possibility. Assume that all digits N_j are equal to M_j for all $j < i$, where i is the position we iterate from left to right. To ensure that M is strictly greater than N, digit M_i must be greater than N_i. We can loop through all digits greater than N_i in $O(10) = O(1)$. At this point, we have fixed a prefix of M.

Next, let's count how many digits we still need to meet the requirement of having k occurrences of 2025. This is done simply: we initialize a counter and scan the prefix from left to right, increasing the counter whenever we find a required digit.

For example, if the prefix is 25032592, the counter would be 5, because this prefix can be interpreted as the number 20252 of length 5. Now that we know how many digits we've already used, we can calculate how many more are needed. Suppose in this example we need 2025 repeated 3 times. Then we need a total of $3 \times 4 = 12$ digits, and since the prefix already provides 5, we need $12 - 5 = 7$ more digits. Clearly, these remaining digits must go at the very end of the number M. All other unfilled digits in M should be zeros.

Using this algorithm, we get several candidate answers, from which we select the smallest valid number. However, if the suffix doesn't have enough space to fit all the required remaining digits, that means the current prefix won't produce a valid candidate, and it can be skipped.

If after this algorithm we don't find any valid candidate, that means M must have a length exactly one digit longer than N. In this case, the answer will be the number of the form $100\dots0020252025\dots2025$.

The described algorithm runs in $O(|N|^2)$, but if we maintain a running count of the 2025 digits while scanning the prefix and observe that we don't need to compare all candidate numbers—only the one with the longest matching prefix—then we can reduce the time complexity to $O(|N|)$.

Problem C. Yet Another Robin Hood Problem

In the first subtask, it is sufficient to notice that it is more advantageous for the Sheriff to rob the most profitable mansions in advance. Accordingly, it is required to sort the array a in non-decreasing order, after which we get a new array, which we denote by b. In this case, the answer to the subtask will be $b_1 + b_2 + \cdots + b_{n-k}$.

In the second subtask, Robin Hood can either burn down the only police station or not do this. In case he burns it down, we get the first subtask. If Robin Hood does not burn the police station, then let us denote by b_1 and b_2 two continuous subsequences of the original array a sorted in non-decreasing order, not containing -1, and by $b_{i,j}$ — the j-th element of array b_i. Accordingly, the answer is:

$$\min_{\max(0,k-|b_2|)\le i\le \min(k,|b_1|)} \max(b_{1,1} + b_{1,2} + \cdots + b_{1,|b_1|-i}, b_{2,1} + b_{2,2} + \cdots + b_{2,|b_2|-(k-i)})$$

The value of this expression can be found using a greedy algorithm.

The third subtask is a direct continuation of the second. In other words, now we have not 2 arrays, but m. Accordingly, let c_i be an array sorted in non-decreasing order, consisting of elements of arrays b_i and b_{i+1}. Suppose Robin Hood burned down police station i. The answer is:

$$\max_{0\le i<m} \min_{\substack{0\le j_l\le |b_l|,\\ j_1+j_2+\cdots+j_m=k}} \max(b_{1,1} + b_{1,2} + \cdots + b_{1,|b_1|-j_1}, b_{2,1} + b_{2,2} + \cdots + b_{2,|b_2|-j_2}, \cdots,$$

$$b_{i-1,1} + b_{i-1,2} + \cdots + b_{i-1,|b_{i-1}|-j_{i-1}}, c_{i,1} + c_{i,2} + \cdots + c_{i,|c_i|-j_i-j_{i+1}}, b_{i+2,1} + b_{i+2,2} + \cdots + b_{i+2,|b_{i+2}|-j_{i+2}}, \cdots,$$

$$b_{m,1} + b_{m,2} + \cdots + b_{m,|b_m|-j_m}).$$

The final value is found by iterating the station that must be burned down and using aforementioned greedy algorithm. In case Robin Hood did not burn down a police station, this formula can be easily adapted to this by replacing the part with c_i with parts with b_i, b_{i+1}.

The fourth and fifth subtasks require a different approach. Note that Robin Hood can rob x or less money if he can rob $x-1$ or less money. Accordingly, if Robin Hood cannot rob x or more money, then he also cannot rob $x+1$ or more money. Accordingly, we can find the desired value using binary search. We claim that the answer lies within the half-interval $[0, 10^9+1)$ (0 money he can rob, while 10^9+1 money he definitely cannot). Then, we simply narrow our interval, after which we get an interval of the form $[x, x+1)$. The answer is x. Now we need to learn how to process data inside the binary search. Suppose we are now iterating through the value x. First, we need to calculate for each b_i the number of mansions j_i that need to be robbed in advance so that for all i the inequality $b_{i,1} + b_{i,2} + \cdots + b_{i,|b_i|-j_i} < x$ holds. Let us denote this value by c_i. If $c_i > k$, then Robin Hood can rob x money. Otherwise, he will have to try to burn down some police station. Note that after burning down the i-th station, only c_i and c_{i+1} change. For them, we need to find the new number of mansions that need to be robbed in advance, after which we again check against k. If after burning down any police station the number of mansions that need to be robbed in advance does not exceed k, then Robin Hood cannot rob x or more money. These subtasks differ only in implementations. In the fourth subtask, one can only store the number of elements in each continuous subsequence, while in the fifth subtask, one must already store the original subsequences in sorted form.

Problem D. Déjà Vu

Subtasks 1 and 2

If $R - L \le 1$, then we simply need to output either A_L or $A_L^{A_R}$. With naive exponentiation, we get a complexity of $O(Q \cdot \max A_i)$, which fits within the constraints when $A_i \le 100$. For larger values of A_i, binary (fast) exponentiation must be used. The time complexity in this case is $O(Q \cdot \log(\max A_i))$.

Subtasks 3 and 4

Using the property

$$a \equiv b \pmod m \Rightarrow a^k \equiv b^k \pmod m$$

we can see that it suffices to iteratively raise the results of previous computations to a power. This leads to a time complexity of $O(Q \cdot N \cdot \max A_i)$ with naive exponentiation and $O(Q \cdot N \cdot \log(\max A_i))$ using binary exponentiation.

Subtask 5

Note that a "left-associative" exponentiation tower can be rewritten as:

$$A_L^{A_{L+1} \cdot A_{L+2} \cdot \ldots \cdot A_R}.$$

If M is a prime number, then by Fermat's Little Theorem, for any A_L coprime with M, the following holds:

$$A_L^n \equiv A_L^{n \pmod{M-1}} \pmod M.$$

If A_L is divisible by M, then the answer is 0. Otherwise, we compute the exponent modulo $M - 1$ and raise A_L to that power using binary exponentiation. To efficiently compute the product over the interval $[L + 1, R]$, we can use a segment tree. The resulting complexity is $O(N \log N + Q(\log N + \log(\max A_i)))$.

Subtask 6

Let $M = p^k$. Then:

- If A_L is divisible by p, then A_L^{32} is divisible by M, since $k \le \log_2(\max A_i) \le \log_2(10^9) < 32$. Therefore, if the product of numbers in the interval $[L+1, R]$ is at least 32, the result is immediately 0. Otherwise, we can compute the power directly.
- If A_L is not divisible by p, then A_L and M are coprime, and we can use the approach from *Subtask 5*, applying Euler's theorem instead of Fermat's.

Subtask 7

Lemma. If $A \equiv B \pmod{M_i}$ for all i, and the moduli $M_1, M_2, \ldots, M_k$ are pairwise coprime, then:

$$A \equiv B \pmod{M_1 \cdot M_2 \cdot \ldots \cdot M_k}.$$

Proof. From $A \equiv B \pmod{M_i}$, it follows that $A - B$ is divisible by each M_i. Since the moduli are pairwise coprime, $A - B$ is also divisible by their product. ■

Let $M = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k}$ is the prime factorization of M. For each prime divisor p_i, consider two cases:

- If A_L is not divisible by p_i, then by Fermat's Little Theorem:

  $$A_L^n \equiv A_L^{n \pmod{p_i - 1}} \pmod{p_i}.$$

  Since Euler's totient function $\varphi(M)$ is divisible by $p_i - 1$, it follows that:

  $$A_L^n \equiv A_L^{C \cdot \varphi(M) + (n \pmod{\varphi(M))}} \bmod p_i$$

  where C is any integer.

- If A_L is divisible by p_i, and since $\varphi(M) \geq \alpha_i$ for all natural numbers, then:

  $$A_L^n \equiv A_L^{\varphi(M)+r} \equiv 0 \pmod{p_i}$$

  when $n \geq \varphi(M)$ and $r \geq 0$. If $n < \varphi(M)$, we can compute the power directly.

Thus, if the product of elements in the interval $[L+1, R]$ exceeds $\varphi(M)$, then:

$$A_L^n \equiv A_L^{\varphi(M)+(n \pmod{\varphi(M)})} \pmod{M}.$$

Problem E. Volatile Words

To solve the first two subtasks, it is sufficient to explicitly append prefixes to the strings for queries of the second type. For queries of the first type, one should iterate over all strings in the range from l to r and count how many of them have the required prefix.

To solve the third subtask, several approaches can be used. One such approach is to build a trie, where each node stores a sorted array of indices of strings in the array that share the prefix corresponding to that trie node. Then, to answer a query, you find the trie node corresponding to the query string. Using binary search, you can find the first index $\geq l$ and the first index $> r$, and their difference gives the number of strings with the required prefix. The total complexity of the algorithm is $O(n + L_s + q + L_v)$.

For the full solution, we reverse all strings in the array and in the queries. Then, queries of the first type become equivalent to finding the number of strings with a given suffix, and queries of the second type simply append a suffix to the end of a string.

We will use polynomial hashes and compute prefix hashes for each string in the array. When adding a new suffix to a string, we can compute the hashes of the new prefixes in $O(|v|)$. This allows us to compute the hash of any suffix of any length in $O(1)$ at any time.

Next, note that the number of distinct lengths of v in queries of the first type is no more than $O(\sqrt{L_v})$. Indeed, assuming otherwise, we would have $L_v > 1 + 2 + \ldots + 2\sqrt{L_v} = \frac{2\sqrt{L_v}(2\sqrt{L_v}+1)}{2} > 2L_v > L_v$ which is a contradiction.

Therefore, we can maintain $O(\sqrt{L_v})$ dictionaries, each mapping hash values of suffixes of a specific length to an ordered set (e.g., a treap with explicit keys) of indices of strings that have such a suffix. In this case, answering a query of the first type reduces to computing the hash of the suffix and finding the difference in order positions of l and $r+1$ (as in subtask 3) in the dictionary corresponding to its length. For queries of the second type, we remove the index from the ordered sets corresponding to the old suffixes and insert it into the sets corresponding to the new suffixes.

The final complexity of the algorithm is $O(L_s + n\sqrt{L_v} + q\sqrt{L_v}\log n)$.