Problem A. Load Distribution

Consider two cases:

- Begimai is reading towards L:
  - Viktor will definitely read all problems from n to s (exclusive) — a total of $(n - s)$.
  - Aidar and Begimai will read the same number of problems, which is $\lceil \frac{s}{2} \rceil$.
- Begimai is reading towards R:
  - Aidar will definitely read all problems from 1 to s (exclusive) — a total of $(s - 1)$.
  - Begimai and Viktor will read the same number of problems, which is $\lceil \frac{n-s+1}{2} \rceil$.

P.S. The notation $\lceil \frac{x}{2} \rceil$ denotes the integer division of x by 2 rounded **up**.

It can be computed in the following ways:

- either using a conditional operator for "divides / does not divide";
- or as $(x \text{ div } 2) + (x \text{ mod } 2)$;
- or as $(x + 1) \text{ div } 2$.

where div denotes integer division, and mod denotes the remainder of the division.

P.P.S. Think about how the formula would look in general for $\lceil \frac{x}{y} \rceil$.

Problem B. What to solve next?

Let's immediately note that all input data were separated by spaces, which allowed either reading element by element (`std::cin` in C++), or splitting the already entered line (`split` in Java, C#, Python, etc.).

The task itself did not present any conceptual or implementation complexity:

- We will store three counters for each task:
  - S_j — the number of teams that solved task j;
  - T_j — the number of teams that attempted to solve task j;
  - F_j — the total number of attempts on task j.
- For the result of each team on task j, we will check the following conditions:
  - Always increase F_j by the specified number of attempts.
  - If the team solved the task, increase the counters S_j and T_j by 1.
  - If the team did not solve the task but made at least one attempt, increase the counter T_j by 1.

Problem C. Integer Overflow

The solution can be divided into two parts:

- Calculate the "minimum" type for each of the variables A, B, $C = A \cdot B$.
- Adapt the type of variable A or B to match the type of C if necessary.

Let's consider the first step — calculating the "minimum" type:

- For variables A and B, this was not difficult — just compare the values with the constants given in the problem statement.
  - Note that in languages like C++, Java, C#, etc., it was necessary to use a 64-bit type for both storing variables A and B and for storing the constants — the boundaries of the types.
  - It is also important to remember that integer constants in such languages are 32-bit by default, so it was necessary to use the suffix `L` in Java / C# or the suffix `LL` in C++.
- In languages with long arithmetic (e.g., Python), the check for variable C was the same as for A and B.
- In languages without long arithmetic (e.g., C++), one could use the assertion:

  From $A \cdot B \leq X$, it follows that $A \leq \lfloor \frac{X}{B} \rfloor$, after which substitute the upper limits for 32-bit and 64-bit types as X.

  Since all values were positive, no additional checks and conditions were required.

Now let's consider the second step — adapting the types of A or B to match the type of C:

- Choose the larger type for A and B.
- Make only that one equal to the type of C, without changing the smaller type.
- Thus, we will satisfy the necessary condition and achieve the optimal sum of bit sizes:
  - if the types of A and B are equal, there is no difference in which one to increase;
  - if the types of A and B are not equal (i.e., 32 and 64 in some order), then:
    - either the type of C is already 64 and nothing will change;
    - or the type of C is 128 — and the sum $32 + 128 + 128$ will be less than $128 + 64 + 128$.

Problem D. TL, ML or OK?

In this problem, it was required to carefully perform calculations according to the formulas described in the statement:

- the number of operations in one iteration was equal to $(2 \cdot q + 5 \cdot k)$;
- the number of integers added in one iteration was equal to k.
- to calculate the total quantities, both values needed to be multiplied by n.

P.S. Of course, it was necessary to use a 64-bit type for calculations (hello to problem C);

Problem E. Final Rankings

The solution can be divided into three subtasks:

- determine the final number of solved problems and total penalty time for each team;
- sort the teams in the correct order;
- calculate the final rankings of the teams.

The first part of the solution is determining the results for each team:

- For each pair (team i, problem j), we will store two values:
  - $AC_{i,j}$ — whether team i has already solved problem j or not;
  - $PN_{i,j}$ — the total penalty time that team i will incur if it solves problem j.
- If the current submission in the log is made for an already solved problem — do nothing.
- If the submission is unsuccessful, increase $PN_{i,j}$ by 20 — the specified penalty time for an incorrect attempt.
- If the submission is successful, increase $PN_{i,j}$ by the number of minutes indicated in the log, and then update the flag $AC_{i,j}$.

The second part of the solution is sorting the teams:

- Since there were at most 10^3 teams, any sorting algorithm would work — both $O(n^2)$ and $O(n \cdot \log n)$.
- The comparator used in the problem is quite standard — lexicographic comparison of tuples in the order:
  - S in descending order;
  - X in ascending order;
  - ID in ascending order.
- In C++ and Python, one could use built-in tuple/array types, for which lexicographic comparison in ascending order is implemented by default (it was necessary to store S with a negative sign).
- More about comparators and sorting in C++, Java, Python can be found in (part of the "Data Structures" course).

The third part of the solution is determining the final rankings:

- The team in position 1 in the sorted order has a rank of 1 by definition.
- To determine the rank of the team in position i in the sorted order:
  - consider the pair (S_i, X_i);
  - compare it for **equality** with the pair (S_{i-1}, X_{i-1}):
    * if the pairs are equal — the rank of the i-th team is the same as the rank of the $(i-1)$-th team;
    * otherwise — the rank is equal to i.
- Ranks could be stored either in a separate array or as part of the tuple from the second part of the solution.

Problem F. Compromise

First, let's note the following fact: the characters in the answer at positions i and $n-i+1$ will always be equal (by the definition of a palindrome) and will depend only on the characters S_i, T_i, S_{n-i+1}, T_{n-i+1}.

For this problem, there were two approaches to determine the optimal character in the answer:

- either iterate through all options;
- or compute it.

The first method of solving is to iterate through all options:

- Note that there are only $A = 26$ characters in the Latin alphabet.
- Iterate through the assumed position in the alphabet for the character in the answer (from 1 to A).
- Calculate the total distance from the quartet of characters in the original strings to the current one (using ASCII codes).
- The overall complexity of this solution will be $O(n \cdot A)$.

The second method of solving is to compute the optimal character:

- Let c_1, c_2, c_3, c_4 be the original quartet of characters in **ascending** order of their positions in the alphabet (ASCII codes).
- It is claimed that any character from the interval $[c_2, c_3]$ will be optimal—we will always use either c_2 or c_3.
- The overall complexity of this solution will be $O(n \cdot k \log k)$, where $k = 4$.

P.S. For languages like Python, Java, C#, etc., it is important to remember that character-wise string concatenation in a loop is not asymptotically efficient—you should use classes like `StringBuilder` or collect the answer in a dynamic array and use methods like `join`.

P.P.S. This problem is a special case of the problem of **minimizing total distance on a line**, which is classical.

It is known that its solution is the **median** of the array—the element that stands in the middle of the array in **sorted order**.

In the case of an even number of elements—any point in the interval between the two medians.

Let's provide a simple proof (though not entirely formal):

- Place the answer on the line in the interval between points x_i and x_{i+1}, denote the total distance as S.
- Move the answer 1 to the left—S will decrease by i (since the answer became closer to the first i points), but will increase by $(n - i)$ (since the answer became further from the last $(n - i)$ points).
- Similarly, consider moving the answer 1 to the right.
- Note that the answer will be in the optimal position if moving either left or right increases S.
- It is easy to see that the optimum is reached when $|i - (n - i)| \leq 1$, from which it follows that $|2 \cdot i - n| \leq 1$—from here it is easy to derive the statements described above.

Problem G. A + B = C

The restriction of 3 on the length of strings SA and SB (and, accordingly, the upper limit of 999 on A and B) allowed solving this problem by simply enumerating all possible options in $O(10^{|SA|+|SB|} \cdot (|SA| + |SB| + |SC|))$:

- We will enumerate all possible pairs A and B that fit in two nested loops according to the number of digits.
- We will calculate the expected value $C = A + B$.
- We will check the correspondence of string masks to fixed numbers and the absence of violations of the bijection "one letter — one digit":
  - We will create two dictionaries — one will store the expected digit for a letter, the other will store the expected letter for a digit;
  - We will iterate through the numbers and check that for the current pair (letter, digit) there were no other correspondences found in the dictionaries;
  - Leading zeros will be handled automatically, as the numbers A, B, and C do not contain them by construction.

Problem H. Parallel Checking

Let $F(d)$ denote the total number of quanta of time with a fixed number of threads d:

$F(d) = \sum \lceil \frac{A_i}{d} \rceil$.

Let $G(d)$ denote the predicate $F(d) \leq T$.

Our goal is to find the minimum d such that $G(d)$ is true.

Note that $F(d) \geq F(d+1)$ for all $d \geq 1$, which implies that $F(d)$ is a monotonically non-increasing function.

From the monotonicity of $F(d)$, it is easy to see that:

- if $G(d)$ is true, then $G(d+1)$ is also true;
- if $G(d)$ is false, then $G(d-1)$ is also false.

This means that the function $G(d)$ is also monotonic, and there exists a D such that:

- for all $0 < d < D$, the predicate $G(d)$ is false;
- for all $D \leq d$, the predicate $G(d)$ is true.

We will use the technique of **binary search on the answer** to find the value of D in $O(\log \max A \cdot n)$.