A: Lawnmower

The high-level idea is that we want to build a list of events, corresponding to both unit squares and lawnmowers available at the shop, and then sweep them, computing the answers for the lawnmowers in the order we encounter them.

What events we should consider? For a lawnmower with blade height b, we can create an event for it with priority b. For a unit square containing grass of height g, there are two values of interest: $\lceil \frac{g}{2} \rceil$, which is when the square becomes *active*, i.e. can be entered at all, and g, when it becomes *uninteresting*, meaning that it no longer contributes to the total amount of grass we mow.

This gives us $2nm+s$ events, which we consider in the order of increasing priority. On the side, we want to maintain a division of the active squares into connected components (but need to consider an extra vertex corresponding to the outside of the lawn); moreover, for each component, we want to store the sum of grass heights of all the interesting squares belonging to it, as well as the number of such interesting squares. This can be done with a standard Disjoint Set Union datastructure, enriched with two extra statistics for each component. When a square becomes active, we possibly join some components; when it becomes uninteresting, we update the values stored in its component. Finally, whenever we reach an event corresponding to the lawnmower, we read-off the sum s and count c for the component containing the auxiliary vertex corresponding to the outside, and compute the answer as $s - c \cdot l_k$, where l_k is the blade height of the lawnmower in question.

The time complexity is dominated by sorting the events, and turns out to be $\mathcal{O}((nm+s)\log(nm+s))$.

B: Bajtastic

It's easy to see that bajtastic words are exactly those starting with either *ajt*, *bajt*, *jt* or *t*. These sets are disjoint; if we denote the size of the alphabet by $|A| = 26$, then these sets of words have sizes $|A|^{n-3}$, $|A|^{n-4}$, $|A|^{n-2}$ and $|A|^{n-1}$, respectively (but if $n < 4$, some of them might be empty).

If we precompute the powers of $|A|$ (remembering that some of them are larger than 10^{18}, in which case we don't care about their exact value), then given k we can easily check which prefix we should use. We then decrease k appropriately, to account for the words we "jumped over", and generate the remaining letters of the word right-to-left (by looking at $k \mod |A|$ to determine the last letter, dividing k by $|A|$, and continuing).

The solution can be implemented in $\mathcal{O}(n)$, which is more than enough to get Accepted.

C: We hate rectangles

If there are two points in any column c, say $A[i][c]$ and $A[j][c]$, then for any other c' the points $A[i][c']$ and $A[j][c']$ cannot appear together, as they would form a rectangle. Therefore, for every pair of rows (i, j) there is at most one column which contains both points of this pair. There are at most $\binom{k}{2}$ pairs in total, and if a column contains $p \geq 2$

points, then it uses up $\binom{p}{2}$ pairs. As $\binom{p}{2} \geq p/2$ for $p \geq 2$, there can be no more than $2\binom{k}{2}$ points in columns with ≥ 2 points. The maximum number of points is then $2022-k+\binom{k}{2}$, and we can achieve it by placing a different pair of points in each of the first $\binom{k}{2}$ columns, and one point in all remaining ones.

D: Minimal Lexicographic Subsequences

Let $S[1..n]$ be the input string, and assume that we are adding letters one by one. Let D_k denote the (current) k-MLS, and $D_k[i]$ be the i-th letter of D_k for $i = 1, 2, \ldots, k$. We will keep all D_k computed throughout the algorithm, so that after each letter answering all the questions is easy. But how to keep all D_k updated after adding the next letter? Let D_k denote the "old" k-MLS (before the j-th letter), and D'_k the new one. We claim that $D'_k[1..k-1] = D_{k-1}$. Indeed, $D'_k[1..k-1]$ cannot contain $S[j]$. If $D'_k[1..k-1] > D_{k-1}$, then $D_{k-1} \cdot S[j]$ would be better than D'_k. There cannot be $D'_k[1..k-1] < D_{k-1}$, as D_{k-1} is minimal in $S[1..j-1]$. Then $D'_k[1..k-1] = D_{k-1}$ and we already know all but one letter of D'_k. The last letter of D'_k can be chosen greedily as the smallest letter between the next-to-last ($D'_k[k-1]$) and the end of S.

To sum up, in every step we compute D'_k by simply taking D_{k-1} and adding one character, a minimal one on some suffix interval of S. We achieve it by keeping all D_k on a vector or deque, and each added letter is computed with a single query to an interval tree in $O(\log n)$. Alternatively, we can preprocess all such suffix queries in $O(n)$ time every time a letter is added to S. The total complexity is either $O(n^2 \log n)$ or $O(n^2)$. Also, the memory complexity is $O(n^2)$ (and we have no idea how to bring it down).

E: We love rectangles

The solution of this problem can be split into two parts: transforming the problem into its equivalent graph form, and then applying some known algorithms on this graph. So what graph should we consider? Take a graph G in which the vertices represent rectangles, and an edge $u-v$ exists if the two rectangles u and v have non-empty intersection. We claim that the answer to the problem is YES if and only if the graph G contains an induced cycle of length at least four.

The proof is somewhat tedious; but we can give an intuitive reason why this claim is true. So suppose we chose a set S of rectangles that splits the plane into at least two parts. Then there exists some point P in the plane which is "surrounded" by rectangles (formally, it belongs to a bounded connected component). We then see, that we can in fact restrict S to some cycle C which "wraps" around point P. Then, by an usual algorithm for finding induced cycles, for any diagonal edge in C we split it into two cycles and pick the one which still "wraps" around P. Eventually we restrict S to just some induced cycle. We finish the argument in one of two ways. We can just notice that such situation cannot occur using only 3 rectangles - so at least 4 are needed. But to give it more rigour, we can notice that the vertical and horizontal lines passing through P must intersect 4 different rectangles from set S. Hence the cycle has length at least 4. The proof in the other direction is even more tedious (and provides little to no intuition), so we will omit it completely.

Hence the algorithm firstly constructs a graph G and then checks if there exists an induced cycle of length ≥ 4 in it. But how do we do that?

This is where we introduce new terminology. First, a graph G is called **chordal** if it does **not** contain an induced cycle of length at least 4. Next, given some graph G and an ordering $v_1, v_2, ..., v_n$ of its vertices, we say that the ordering is **perfect**, if for any vertex v_k the set of its left neighbours (that is, neighbours v_j with $j < k$) forms a clique. Lastly, we introduce the **Lexicographic BFS Algorithm**. Here is how it works: at each step, pick any unvisited vertex v which has **lexicographically the smallest set of visited neighbours** – and then visit it, putting it on the end of the resulting order. We repeat this step until all the vertices have been visited (just as a side note - in Lexicographic BFS given two sets of vertices $A \subset B$ with $A \neq B$, the bigger set B is considered smaller in lexicographic order). Using proper optimizations, we can run Lexicographic BFS in time $O(N^2)$.

But how are these concepts related? Here is how: it turns out that a graph G is chordal if and only if there exists a perfect ordering of its vertices. Furthermore, to check if such an ordering exists, we can find it by running a Lexicographic BFS. So to solve our problem we take the graph G constructed on rectangles, and run Lexicographic BFS on it. If the produced ordering is perfect (we can check it in $O(N^2)$), then the graph is chordal and the answer to our original problem is NO. Otherwise, if the ordering is not perfect, the graph is not chordal, and using an additional algorithm we are able to find an induced cycle of length ≥ 4. Such cycle forms a set of rectangles, which splits the plane in exactly 2 connected components.

F: Departures

Let's start with a slightly modified problem. Suppose that there are no trains running between weeks. We identify each train as an interval, whose endpoints are the departure and arrival times. Two trains may belong to the same group if and only if the corresponding intervals do not nest, i.e. one of them does not start before and ends after the other. Let's sort all the intervals (first by the beginnings and then by the ends). Let's create a list of empty groups to which we will assign intervals, so that each group does not contain a nested pair of intervals. For each group, we maintain the farthest end that is reached by any of the intervals in that group. We go through the sorted list of intervals one by one and append an interval into the first group (we use binary search to find such a group), which farthest end is at most as far as the end of the considered interval. Note that with such assignment, no two intervals in a group can nest. Also note that we can use binary search, because we always add intervals to the first matching group, which means that the ends of the groups form a descending sequence. It can be proved that this group assignment uses the minimal number of groups. We will omit this proof for now and return to it at the end.

Now let's return to the original problem. If the train is running between weeks, then let's consider it twice, as if it was running at the beginning and at the end of the week. Now we could use the same approach as before. However, there might be a problem. We need to prove that every interval which was copied will be assigned to the same group.

Let's observe that in the algorithm we'll start by assigning all copied intervals into groups (they start before the actual week). Then we'll move on to the next intervals until we finally start hitting second copies of the intervals. Note that they are long (sticking out beyond a week). Let's consider what influences the assignment of such an interval. It only depends on the intervals which ends reach farther. However, only other, previously

occurring, copied intervals can affect the assignment, which means that the same intervals as at the beginning have an impact on the location of a given interval, i.e. we will assign them to the same group as at the beginning.

Let's return to the previous statement and prove it. We want to show that presented algorithm indeed uses minimal number of groups. Let's call the process of adding intervals with the same starting endpoint a *phase*. Suppose we've just added a first interval to a k-th group. We've done it because the $k-1$-th group had an interval that was added in one of the previous phases and reaches farther. And that interval was added because of some other interval in the $k-2$-th group and so on. This means that there are k intervals which are pairwise nested. Because of that there cannot exist an assignment into less than k groups. This proves that the presented algorithm uses minimal number of groups.

G: Crystal skulls

The solution is based on the idea of splitting the whole map into regions, and then solving the problem in each region independently. Furthermore, as we will see, each of these regions will in fact be a rectangular strip of width equal to 1 or 2, and maximal height. Lastly, we will traverse these rectangles in a sequential order, always starting in one corner and ending on the opposite.

To put our ideas into practice, we will firstly have to determine the exact partition of map into rectangles. So assume the map has dimensions NxM. Depending on the (surprisingly crucial) value of $m \mod 4$, we will split the map into the following strips:

$m \equiv 0$ Two rectangles of width 1 on both sides of map, and the rest of space is filled with rectangles of width 2.

$m \equiv 1$ One rectangle of width 1 and the rest of space is filled with rectangles of width 2.

$m \equiv 2$ Just the rectangles of width 2.

$m \equiv 3$ In this case, we in fact cut the right-most rectangle of width 1 out of the map, bringing it down to case $m \equiv 2 \mod 4$

This way we always split the map into an odd amount of rectangles, thus ensuring that we start at position $(1,1)$ and end on position (n,m).

Now we just have to figure out how to traverse each of these rectangles. So we note that in our construction we ensured, that each rectangle of width 1 will always be entirely filled with hallways. That's why we will only be concerned with the case of width 2. Here, we only move in one vertical direction, changing the column to collect a treasure/omit a trap whenever necessary. Since all the treasures/traps are sufficiently far away from each other, we are guaranteed to never step on the same field twice.