Subtask 1

Let `dp[i][j]` be the minimum expenditure to arrive in Planet i at time j. Run Dijkstra's Algorithm to obtain the answer. Time complexity is $O(MT \log N)$, where $T = \max_{1 \le i \le M} b_i$.

Subtask 2

Method 1: Prefix Min

Let's define `dist[i]` as the minimum expenditure to finish the train route i. Sort all train routes by increasing b_i and we calculate `dist` in that order. `dist[i]` can be transited from the minimum `dist` value of any other train route which arrives in Planet x_i not later than time a_i. To obtain that value, we can create N arrays, each containing the `dist` values of all train routes ending at a particular planet. Within each array, train routes can again be sorted by increasing b_i, so each query corresponds to a prefix min on `dist` values in the array. We can simply insert each newly calculated `dist` value at the back of the array for Planet y_i and calculate the prefix min at the same time (note that prefix min at earlier positions remains unchanged). In each query, we can use a binary search to find the corresponding prefix to consider. Time complexity is $O(M \log M)$.

Method 2: Dijkstra

Construct a new graph. Each node is now defined as [`current planet`, `current time`]. For each edge in the original graph, we add an edge with the same weight (i.e., c_i) from node [x_i, a_i] to node [y_i, b_i] in the new graph. Hence, we have at most $2m$ nodes. Furthermore, for all nodes of the same `current planet`, we add free (i.e., weight = 0) edges such that they form a directed linear graph of increasing `current time`. For example, if we have train routes arriving/departing in Planet `3` at time 1, 2, 5 only, then we add an edge from [`3`, `1`] to [`3`, `2`] and another edge from [`3`, `2`] to [`3`, `5`]. We can then run Dijkstra on the new graph, where the starting node is [`0`, `0`] and the answer will be the minimum value among shortest distances to any nodes associated with Planet $N - 1$. The time complexity is $O((M + N) \cdot \log M)$.

Subtask 3: meals are disjoint.

Method 2 from Subtask 2 can be improved by modifying the states used in the Dijkstra's Algorithm. For each node in the new graph mentioned above, we introduce another `0/1` state to represent whether the meal covering `current time` has been taken. This meal should be unique under this subtask. Particularly, `1` is assigned if no meal covers the time. The greedy way to take meals is (1) if we are on a train, eat the free meal, (2) otherwise we procrastinate the meal. (2) works as all paid meals must be taken between train routes, so the exact time to take the meal doesn't matter (we would be in the same planet throughout the possible time interval of the meal anyway), meaning that we could just take the meal as late as possible.
From here, we can figure out how to do the transition. For edges corresponding to train routes, we always transit to state `1` since we can always have meals for free on the train. For free edges, we transit to `1` if no meal covers the `current time` of the end node and otherwise transit to `0`. Note that a cost of transition is generated from all meals that happen strictly between the two times at the endpoints; If we are transiting from state 0, there's one extra meal to count if the meal covering the `current time` of the starting node does not last till the `current time` of the end node. Time complexity is $O(M\log(MW))$.

Full solution

Process all train routes in increasing a_i. For each train route i, we store a dynamic `dist` value: the minimum expenditure to finish i and finish all meals that have to be taken before a_{cur}, where cur is the index of train route that's currently being processed. If we store `dist` in a similar way as described in **Method 1 from Subtask 2** (i.e., routes ending at the same planet are stored in one array by increasing b_i), `dist` values can be again obtained from a prefix min. In fact, we can avoid finding a prefix by only appending `dist` values of train routes t which satisfy $b_t \leq a_{cur}$. Appending `dist` values can be implemented with a priority queue over processed trains, sorted by b_i. Then, `dist` values can be obtained from the minimum across the array storing train routes ending at x_{cur} plus c_{cur}.
The next tricky part is how to update `dist` values using meals. Meals that should be counted towards `dist` values satisfy $r_j < a_{cur}$. Since train routes are processed by increasing a_i, we should sort meals by increasing r_j and use them to update all `dist` when appropriate. Note that each meal updates a particular prefix for each `dist` array -- this can't be implemented

efficiently. To optimize this, we treat each array as a monotonic queue, where the `dist` values must be strictly increasing, as meals always worsen a prefix. Then, to obtain `dist` values, we only need to query the first value from the corresponding monotonic queue. But how to handle the updates? For each route p in Planet u's monotonic queue, we can predict when its `dist` value will exceed that of the next route q: Let val_p and val_q be their **initial** `dist` values (i.e., not updated by meals strictly after the train routes), and q will be better than p once $K = \lceil \frac{val_q - val_p}{c_u} \rceil$ meals that satisfy $b_p < l_j \leq b_q$ are processed. The exact meal where the overtaking happen can be identified by finding the Kth largest r_j over a range on meals sorted by l_j. In fact, instead of finding the Kth largest r_j, it might be easier to just give each meal an index based on increasing r_j and find the Kth largest index. Range Kth can be implemented by "walking" on two persistent segment trees, where each persistent segment tree corresponds to a prefix in meals, and its node stores the occurances of indices in a certain range. To find the Kth largest in the range $[L, R]$, We start with two pointers from the roots which corresponds to the prefix of $L-1$ and R respectively. The two pointers should always correspond to the same range in indices, and we can calculate the occurances of indices in the shared range over $[L, R]$ by taking the difference between their corresponding values stored. This allows us to determine whether to go left or right (simultaneously for both pointers) when travelling to a lower layer, and the "walking" stops when the two pointers both arrive at nodes for just one index.
Time complexity is $O(M \log(MW))$, possibly with considerable constant terms.