Assumptions:

1.Based on the example, I have assumed that the coordinates cannot be fractional, for ex 3.65,4.50 is not a valid coordinate. If floating point coordinates are allowed, then it is just a matter of changing the datatype of coordinates to float instead of int

2. I have assumed that each event has a set of tickets where each ticket has a price. So, I have placed the tickets in a priority queue sorted by their price.

The solution is divided into three files:

1. Event.java:
   This class is the blueprint of all events in the solution. All the events that will be created will be objects of this class.
   It has a unique id that is incremented by 1 each time a new object is created
   It holds the set of coordinates for each event.
   It holds a set of tickets for the event which is in the form of priority queue where each ticket has a price, this is done to get the lowest price at the head of the queue and if the program is extended where a user buys the ticket, the head can be removed, and the new head will still hold the least price of the remaining.
   For this problem, we must initialize a set of coordinates which are unique for each event, to check that we have a set of all existing x and y coordinates so that no two coordinates are same.

   There is a initializeSeeds() method to randomly initialize an event
   There is a method getDist() to get the distance between this event and a set of coordinates through Manhattan distance.

2. Checker.java
   This class is used as a comparator interface implementation to sort the events based on their distance from the user input coordinates

3. RunRecommender.java
   This is the class which has the main function to be executed. It randomly initializes a random number of seed events, asks the user for coordinates and displays the nearest ones with their lowest ticket price.

   All functionalities have comments detailing their uses.

   To execute, just execute RunRecommender.java which will prompt you for input coordinates , on entering it , you get the desired output.

   Sample runs:

```
Please input coordinates
5,5
Closest events to 5,5
Event 23 - 9.49, Distance 2.0
Event 10 - 13.20, Distance 3.0
Event 17 - 40.62, Distance 4.0
Event 2 - 21.87, Distance 5.0
Event 5 - 4.80, Distance 5.0


Please input coordinates
-10,4
Closest events to -10,4
Event 6 - 6.60, Distance 2.0
Event 18 - 19.26, Distance 2.0
Event 11 - 2.16, Distance 2.0
Event 3 - 6.51, Distance 3.0
Event 5 - 295.65, Distance 6.0
```

If multiple events at same location need to be supported

If multiple events need to be supported at same location, we could create a class called "Location" which would have x coordinates,y-coordinates as its properties and contain a list/arraylist of Event Objects (Objects of class Event).

Location class would now contain the method getDist() which would return the Manhattan distance between a point and the location. We would also need to change the Checker class to sort Locations instead of Events.
On entering user location, all Location objects in the database would be kept in a priority queue sorted based on their distances (just like we currently do with Event objects). If we need the 5 closest events, we pick the head of the priority queue which would hold the closes location and process and display all the events in that location with its cheapest ticket (from the arraylist of Event Objects which is a property in Location Object). We keep a count of the number of events displayed, if it is less than required, we pick the next closest location.

We can even sort the lowest ticket prices of all the events in a location and display the ones with the least price. For example, if the nearest location has 10 events and we want to display five, we can sort the cheapest price of all events and display the 5 least expensive ones. This can be done by a priority queue as well

If the world size is larger

If the coordinates can range beyond -10 to 10, we have the possibility of getting a lot of events. We need to optimize our code in that case.

Instead of calculating the distance for all the events in the database, we can do an optimized search to generate a candidate set of events which are likely to contain the closest events.
We can arrange all events in a sorted order based on their x coordinate and do a binary search to get the possible events whose x coordinate doesn't differ by a certain threshold, say 20 points from the x coordinate of the user input location.
We do the same and get a set of locations for y coordinates and take an intersection of these points to generate the candidate set. If the candidate set has less than the required values we increase the threshold.

Now, we do the sorting on only the candidate set which is significantly less costly than sorting the entire database of events.