**Problem Name:** **Cultural Fest Seating Plan**

**Problem Statement:**

NIT Srinagar is hosting its annual Cultural Fest, and the organizers need to arrange seats for the event. The seats are divided into sections, and each section has a specific number of seats. The goal is to arrange the seats such that the number of VIP guests and regular guests in each section is as balanced as possible. If it's not possible to achieve a balanced arrangement, the organizers will have to declare the arrangement "not possible".

Brute Force Method: To tackle the Cultural Fest Sitting Seat Plan problem, the brute force approach exhaustively explores all potential seat arrangements to achieve a balanced distribution of VIP and regular guests across sections. Here's a concise breakdown of the method:

* **Total Seats Calculation:** First, calculate the total number of seats by summing up the seats in each section.
* **Even Check:** If the total number of seats is odd, it's impossible to achieve a balanced arrangement, so immediately return "not possible".
* **Target Seats Calculation**: Determine the target number of seats for each side, which is half of the total seats.
* **Iterative Approach**: Utilize a bitmask approach to systematically iterate through all possible combinations of seat assignments to the two sides.
  + For each combination, compute the sum of seats assigned to one side.
* **Balance Check**: If any combination yields a sum equal to the target seats, it signifies a balanced arrangement, warranting a return of "balanced".
* **Outcome Determination:** If no balanced arrangement is identified across all combinations, conclude that achieving balance is not feasible and return "not possible".

**Time Complexity :**

 The brute force method iterates through all possible combinations of assigning seats to two sides.

 For each combination, it calculates the sum of seats on one side by iterating through the sections, which takes O(n) time.

* Since there are 2^n combinations (where n is the number of sections), the overall time complexity is O(2^n \* n).

**Space Complexity:**

* The space complexity is primarily determined by the input size (the number of sections) and the additional space required for variables.
* Additional space is required for variables like totalSeats, targetSeats, mask, and sum, all of which are constant in size, i.e., O(1).
* Therefore, the space complexity is dominated by the input size, which is O(n).

**Implementation:**

 The assignSeats function takes the number of sections (n) and an array of seats in each section (seats) as input.

 It calculates the total number of seats and checks if it's even. If not, it immediately returns "not possible".

 Then, it calculates the target number of seats for each side, which is half of the total seats.

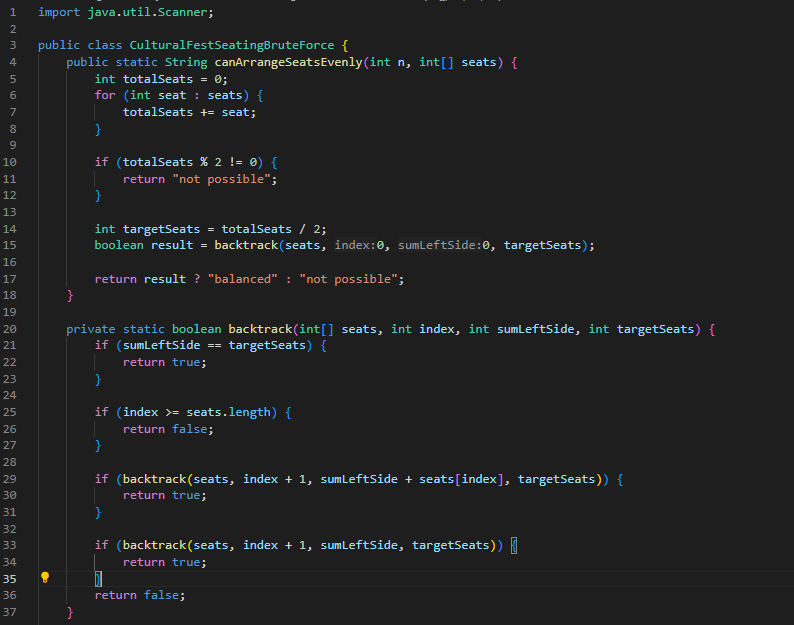
 Next, it iterates through all possible combinations of assigning seats to two sides using a bitmask approach.

 For each combination, it calculates the sum of seats on one side.

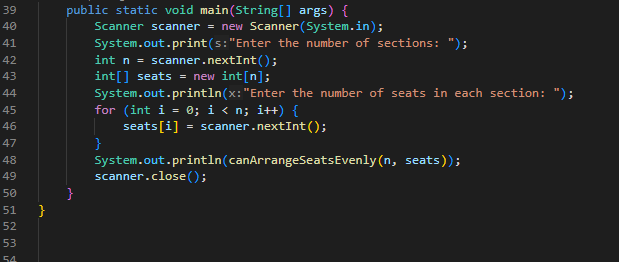
 If any combination results in a sum equal to the target, it returns "balanced"; otherwise, it returns "not possible".

 In the main method, it takes input from the user for the number of sections and the number of seats in each section, and then calls the assignSeats function to determine the possibility of arranging the seating plan.

**Code:**



.



**Code Explanation:**

 ‘**canArrangeSeatsEvenly ‘ Function**:

* **Inputs**: n (number of sections) and seats (an array representing the number of seats in each section).
* **Output**: Returns a string indicating whether it's possible to arrange the seats evenly.
* **Explanation**:
  + Calculates the total number of seats by summing up the elements of the seats array.
  + Checks if the total number of seats is odd. If it's odd, it immediately returns "not possible" since an even distribution won't be feasible.
  + Calculates the target number of seats for each side, which is half of the total seats.
  + Calls the backtrack function with the initial parameters and the target number of seats.
  + Returns "balanced" if the backtrack function returns true, indicating that a balanced arrangement is possible. Otherwise, returns "not possible".

 **backtrack Function**:

* **Inputs**: seats (an array representing the number of seats in each section), index (current index being considered), sumLeftSide (sum of seats assigned to the left side), and targetSeats (target number of seats for each side).
* **Output**: Returns a boolean value indicating whether a balanced arrangement is possible.
* **Explanation**:
  + Base Cases:
    - If the sumLeftSide equals the targetSeats, it means the left side has been assigned the target number of seats, so it returns true.
    - If the index exceeds the length of the seats array, it means all sections have been considered, so it returns false.
  + Recursive Calls:
    - Tries two possibilities at each step:
      * Includes the current section's seats in the left side's sum and recursively calls backtrack with the updated parameters.
      * Excludes the current section's seats from the left side's sum and recursively calls backtrack with the updated parameters.
    - If either recursive call returns true, indicating that a balanced arrangement is possible, it returns true. Otherwise, it returns false.

 **main Method**:

* Takes input from the user for the number of sections and the number of seats in each section.
* Calls the canArrangeSeatsEvenly function with the provided inputs and prints the result.

**Optimized Solution: Dynamic programming**

 **Dynamic Programming Approach**:

* In dynamic programming, we optimize solutions by breaking down the problem into smaller subproblems and storing their solutions to avoid redundant computations.
* Here, we use dynamic programming to solve the problem of evenly arranging seats across sections by efficiently determining if it's possible to form subsets of seats that sum up to half of the total seats.
* The boolean array dp serves as a memorization table, where dp[j] represents whether it's possible to form a subset with a sum of j seats.
* By iteratively updating the dp array based on the seat configurations in each section, we determine the feasibility of arranging the seats evenly.

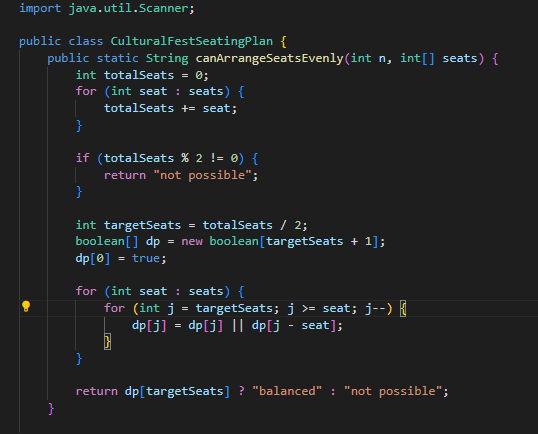
 **Time Complexity**:

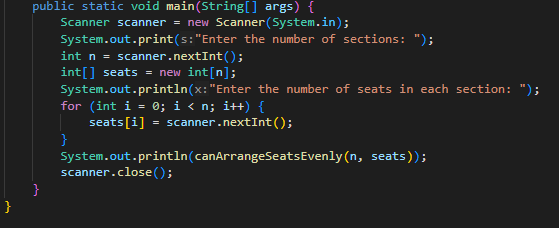
* The time complexity analysis involves understanding how the algorithm's execution time grows with the size of the input.
* In this DP approach:
  + The outer loop iterates through each section (n iterations), which takes linear time, O(n).
  + Within each iteration of the outer loop, the inner loop iterates backwards from targetSeats to seat (the number of seats in the current section).
  + In the worst case, where each section has a significant number of seats, the inner loop runs targetSeats - seat times for each section.
  + Therefore, the total number of iterations across all sections is proportional to n \* targetSeats, resulting in a time complexity of O(n \* targetSeats).

 **Space Complexity**:

* The space complexity analysis focuses on the additional memory required by the algorithm, apart from the input.
* Here, the space complexity is primarily determined by the size of the dp array.
* The size of the dp array is targetSeats + 1, where targetSeats is half of the total number of seats.
* As the size of the dp array is directly related to the target number of seats, the space complexity is O(targetSeats).

**Code:**

****

****

**Code Explanation:**

 **canArrangeSeatsEvenly method**:

* This method takes two parameters: n, the number of sections, and seats, an array representing the number of seats in each section.
* It calculates the totalSeats by summing up all the elements in the seats array using a loop.
* If the total number of seats is odd (totalSeats % 2 != 0), it returns "not possible" immediately since it's impossible to evenly arrange an odd number of seats.
* It calculates the targetSeats, which is half of the totalSeats.
* It initializes a boolean array dp of size targetSeats + 1 to store whether it's possible to form subsets of seats summing up to each value from 0 to targetSeats.
* The dp[0] is set to true since it's always possible to form a subset summing up to 0 seats.
* It then iterates through the seats array and for each seat count seat, and for each value j from targetSeats down to seat, it updates dp[j] to true if either dp[j] or dp[j - seat] is true. This step efficiently determines all possible subset sums of seats that can be formed.
* Finally, it returns "balanced" if dp[targetSeats] is true, indicating that it's possible to evenly arrange the seats, and "not possible" otherwise.

 **main method**:

* The main method handles user input and program execution.
* It creates a Scanner object to read user input from the console.
* It prompts the user to enter the number of sections (n) and the number of seats in each section.
* It reads the input values into the n and seats variables.
* It then calls the canArrangeSeatsEvenly method with the provided inputs and prints the returned result.
* Finally, it closes the Scanner object to release system resources.

**Input :**

* The user is prompted to enter the number of sections.
* Then, the user is asked to input the number of seats in each section, separated by spaces.

**Output :**

* The program prints either "balanced" if it's possible to arrange the seats evenly between two sides, or "not possible" if it's not achievable.

**One Compiler Link:**

* [Cultural Fest Seating Plan](https://onecompiler.com/java/42dgtkmq9)
* [Cultural Fest Seating Plan Brute Force](https://onecompiler.com/java/42dh99q9h)