# Movie Theater Seating System Report

Jingchen Ye

October 19, 2017

## 1 Assumption

The system receives all the booking requests and handle them together, i.e. the system knows all the data in advance and then finds an optimal solution. If requested seats are beyond movie theater's capacity, the system cuts off at the request that leads to overflow, and refuse all the remaining requests.

## 2 Customer Satisfaction Model

Every customer has an expectation that all the seats he/she reserved are contiguous. However, when people request a large number of seats, they don't have the desire as strong as those who request fewer seats. Moreover, if two customers request the same number of seats, the one reserved earlier (with smaller reservation id) is guaranteed to have higher satisfaction.

Based upon the aforementioned facts, I build a quantitative model to characterize customer satisfaction. Let $a_i$ be the number of seats in the reservation request id R$i$.
1. If $a_i > 20$ (the number of seats in each row), the system definitely needs to break $a_i$ into $a_i/20$ full rows and a new request with $a'_i = a_i\%20$. Therefore no penalty will be added to the total customer satisfaction in this case.
2. I assign 1 points to each of the reservation request, and subtract

$$\frac{\text{the number of external seats - 2}}{\text{the total number of sides}}$$

from the total score if the algorithm breaks the seats of a single reservation into several groups.

## 3 Seat Assigning Algorithm and Data Structures

Since obtaining a perfect assigning solution is NP-hard, I choose to use greedy algorithm to maximize the score of my customer satisfaction model, and also theater utilization.

### 3.1 Algorithm

Step 1: Pair two sum
Firstly I build a TreeMap and pair two requests where $\Sigma a_i = 20$. During this

process, the algorithm can also identify some “giant” requests which are larger than 20 and transfer them to smaller ones with $a'_i = a_i\%20$ and $a_i/20$ full rows.

Step 2: Break large requests
Next steps, I pick up all the large requests R$i$ where $a_i$ plus the smallest $a_j$ among all requests is still larger than 20, i.e. $a_i + a_j > 20$, where $a_j = \min a$. Breaking these large requests doesn't hurt the optimality of algorithm. I choose to break $a_i = a_i^{(1)} + a_i^{(2)}$ such that $a_i^{(1)} + a_k = 20$, where $a_k$ is the nearest request to 10 (half of 20), i.e. $a_k = \min|a_m - 10|$, $\forall m \in \{1, 2, \cdots, n\}$, where $n$ is the number of requests. This could guarantee neither $a_i^{(1)}$ or $a_i^{(2)}$ become another large request that cannot be paired.

Step 3: Subset sum
Then I find out all possible subsets in which all the requests' $a_i$ sum to 20 by taking advantage of dynamic programming. Note that these subsets could have overlaps, and therefore I need to identify the ones with priority. Based on my customer satisfaction model, choosing to break the requests with smaller $a_i$ into parts will lead to larger penalty than breaking those with larger $a_i$. Hence I need to assign contiguous seats to smaller requests as early as possible so that they won't be broken into parts in the later stages. Therefore I sort these subsets with such a comparator that the larger size the subset has, the higher priority it will obtain. Because the sum of all requests' $a_i$ in a subset is fixed to be 20, if the subset has more requests, it should contain smaller requests. So that I can fulfil the requests with small $a_i$ first. In case that two subsets have the same size, the one with smaller largest request has higher priority.

Step 4: Greedy algorithm
So far I have finished all perfect pairs/subsets. Starting from this point, I have to break large requests with penalties involved in order to meet a single row. In order to minimize the penalties, I tend to fill a row with sorted requests in ascending order, all the way until current occupancy + next request's $a_i > 20$. It need a request broken here. I fetch the largest request from the end of requests, and break it into two parts: one for current row, the other for next row. Note that although customers with large reservation request have prepared for potential noncontinuous seats, they'll be unhappy if their seats are divided into multiple groups. Hence once I break a request once to meet other requests, I will insert its remaining part to a new row immediately to ensure it won't be broken again. Another point is that I try to avoid breaking a large request to one single seat and another part. This can be achieved by checking if the capacity allows discarding the empty slots of current row. In this way, the algorithm handles all requests to fill the remaining rows.

## 3.2 Example

Assume we have requests
R001 43, R002 59, R003 17, R004 4, R005 15, R006 11, R007 13, R008 8, R009 7, R010 2, R011 3, R012 2, R013 3, R014 6, R015 14.

Pre-processing:

Since the first 14 requests sum to 193, so the system will reject the requests after R015(inclusive).

1. Find two sum pairs and break giant requests
New rows:
Row 1: (R001) 20;
Row 2: (R001) 20;
Row 3: (R002) 20;
Row 4: (R002) 20;
Row 5: (R001) 3; (R003) 17;
Row 6: (R007) 13, (R009) 7;

Remaining requests:
(R002) 19, (R004) 4, (R005) 15, (R006) 11, (R008) 8, (R010) 2, (R011) 3, (R012) 2, (R013) 3, (R014) 6, (R015) 10.

2. Break large requests:
The smallest remaining request is (R010) 2. For (R002) 19, $19 + 2 = 21 > 20$, it needs to be broken. The nearest request to 10 is (R006) 11. Therefore I insert a new row
Row 7: (R006) 11 (R002) 9
and a new request as (R002) 10 into the request map.

Remaining requests:
(R002) 10, (R004) 4, (R005) 15, (R008) 8, (R010) 2, (R011) 3, (R012) 2, (R013) 3, (R014) 6, (R015) 10.

3. Find subsets that sum to 20
Here is sorted subsets.
$[6, 4, 3, 3, 2, 2]$
$[8, 4, 3, 3, 2]$
$[10, 3, 3, 2, 2]$
$[8, 6, 3, 3]$
$[8, 6, 4, 2]$
$[10, 4, 3, 3]$
$[10, 6, 2, 2]$
$[10, 8, 2]$
$[15, 3, 2]$

The algorithm will choose non-overlap subsets in this order to make a new row.
Row 8: (R014) 6 (R004) 4 (R011) 3 (R013) 3 (R010) 2 (R012) 2

Remaining requests:
(R008) 8 (R002) 10 (R005) 15

4. Greedy algorithm
The algorithm traverse the TreeMap to make a new row.
Row 9: (R008) 8 (R002) 10 (R005) 2
Row 10: (R005) 13

The algorithm only break two requests, and there is no penalty for one of them. The total penalty for this input is $\frac{2}{13}$.

### 3.3 Time Complexity Analysis

1. The time complexity of paring two requests where $a_i + a_j = 20$ is $O(n)$ and constructing the TreeMap is $O(n \lg n)$, where $n$ is number of reservation requests.
2. Breaking a large request and inserting the remaining part cost $O(\lg n)$, so the total cost is less than $O(n \lg n)$, depending on the number of large requests that need to be broken.
3. Finding subset sum and tracking the path using dynamic programming is $O(mn)$, where $m = 20$, and sorting the subsets costs $O(k \lg k)$, where $k$ is number of subsets. It can be ignored compared to $O(mn)$ since $k \ll n$.
4. Greedy algorithm costs at most $O(n)$.

Therefore the algorithm runs in $O(n \lg n)$ in total.

### 3.4 Data Structures

1. In "AssignerController", I build a TreeMap to store the reservation requests. The key of the map is the number of seats each reservation requests and the value of the map is a min heap sorted by reservation id. I use Priority Queue to implement the heap. The reason why I use a min heap is that I can guarantee the request with smaller reservation id can always be fetched earlier.

2. Then I use ArrayList<Integer> to store the sorted keys in TreeMap as the input of dynamic programming. I use a two-dimensional array to record the sum that subsets can achieve, and ArrayList<Integer> to track the composition of subsets. A HashSet which stores subsets in List<Integer> is utilized to avoid the duplicate subsets brought by duplicate $a_i$. Since input keys of dynamic programming are in ascending order, and I track back to find the paths, the elements in each subset will always be descending order. This provides convenience to sort these subsets.

3. In "Layout", I use HashMap to store the seats' id that are assigned to each request. And in "Row", List<Request> is employed to represent the requests that each row possesses.

### 3.5 Summary

Thanks to greedy algorithm and dynamic programming, I abandon the backtracking method and ensure the algorithm runs in a remarkable $O(n \lg n)$ time, but meanwhile give an pretty good seat assignment solution under my customer satisfaction model.

## 4 Object-Oriented Design

See Figure. 1.

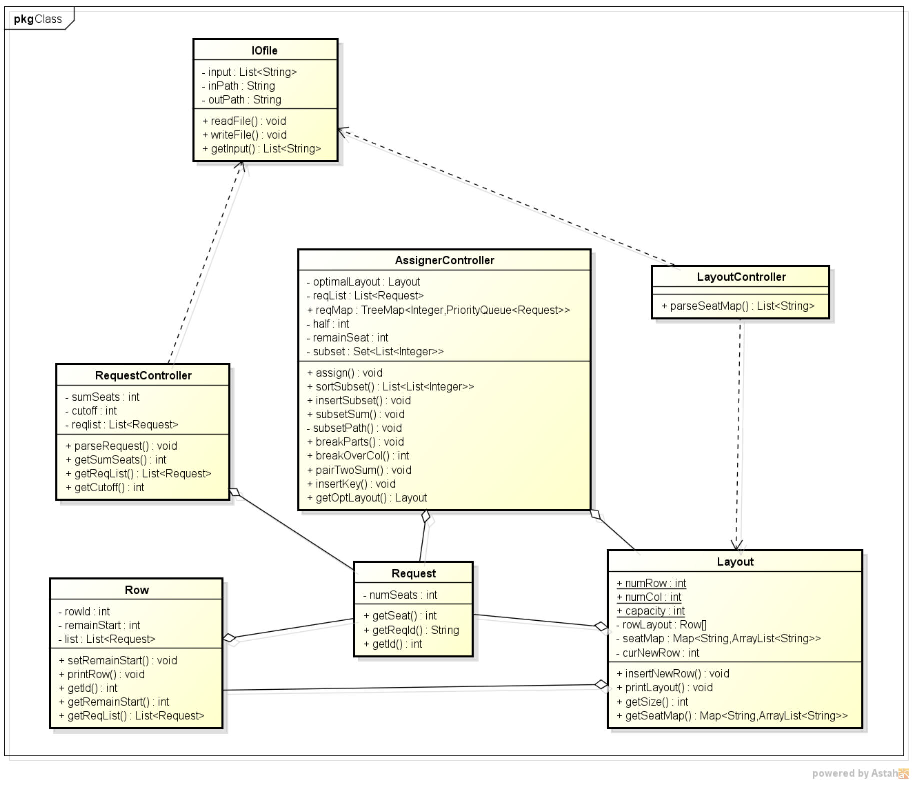

Figure 1: Class Diagram