Array A is strictly increasing because Adam is driving in one direction, then binary search could be used here.

By using binary search but finding the closest value to b. Whenever we need to search in the left side, then we check whether the target value (b) is greater then previous mid pole. If it is, meaning the target is inbetween current mid pole and previous mid pole, then compare the differences of those (mid poles) with the target value to get the closest distance among them. Do the opposite checking whenver we need to search in the right side.

Using the value from the algorithm as the answer.

Overall time complexity is O(log n) for using binary serach in n\_th length array, because the traversing part is the same as binary search with extra O(1) condition checking.

Binary serach could be used here as explained in 9(a).

Since each i\_th night, Adam would be in A[i] motel and Claire would also be in C[i] as well. And the distance between the two are |d - C[i] - A[i]| because Claire is driving from Adam's destination with distance d.

Using similar algorithm as 9(a) but the value pole we are using is the distance between the two instead of value in Array A.

Overall time complexity is O(log n) for using binary serach in n\_th length array, because the traversing part is the same as binary search with extra O(1) condition checking.

Whenever the sum of their positions are greater than n. (sum of number of moves they require to reach square n need to be less than n).

Because only one of the ghosts could move at a time, if we need both of them to reach square n, then their number of moves shouldn't take longer than Mrs. Pacman to reach any of them. And the order of moves doesn't matter as long as we move the one closest to Mrs. Pacman first.

Iterating an array A from the last element and check for how many moves the ghost require to reach square n (n - A[i]) and add the value as the sum of moves require for that many ghosts. Whenever the sum of moves exceed n, meaning Mrs. Pacman would be able to reach all other ghosts by now, causing k - i to be the maximum number of ghosts that could be saved.

The way computing the sum of number of moves is using 10(a) logic. And this is the maximum number of ghosts because we prioritize the ghost that require the least move first, so that the other ghosts would have more moves to spend on before got reached by Mrs. Pacman.

Overall time complexity is O(k) for iterating a k\_th length array.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 2 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |

In this case, the provided algorithm will choose the pad with 2 flies because it is the highest value in row 1, but then would return 0 when it couldn't find the next pad to jump to. However, the answer could be 3 for using the first pad then jump in the first column until reaching the last row.

**Subproblems**: for each 1 <= i <= r and 1 <= j <= c, let P(i,j) be the problem of determining opt(i,j), the maximum number of flies eaten via possible path to the pad.

**Recurrence**: for i > 0, and 1 <= j <= c,

opt(i,j) = max{opt[i-1,k] | j <= A[i-1,k] - k && A[i-1,k] + k <= j, 1 <= k <= c} + A[i,j]

opt(i,j) = 0 when max{opt[i-1,k] | j <= A[i-1,k] - k && A[i-1,k] + k <= j, 1 <= k <= c} = 0

Interpreting getting the maximum flies eaten among possible paths and 0 whenever no existing possible path to reach the cell.

Since opt(i,j) depends on all columns of row i-1, we then iterate it row by row (increasing i ties by increasing j).

**Base cases**:

if i = 1, opt(1,j) = A[1,j] for the first row that has no previous row's column to depend on.

Overall time complexity is O(rc^2), for iterating through a 2D-array with r row and c column, and for each cell, we find the maximum value among previous row's column in maximum of c column.

Start by constructing a graph, having a sink node t. For each station, having new vertex connectd by an edge with capacity of A[i] - B[i] from the sink node to the station vertex. Then for each train, having an edge between station vertices X[i] and Y[i] with capacity of P[i] (undirected). Then having a source node s with edges point to every station with capacity of infinity.

The sum of capacity of edges from all stations to the sink node is the number of available spaces.

Getting the maximum flow (Edmonds-Karp algorithm) would give us the number of spaces used. Interpreting the maximum number of passengers from all trains that could get to any stations.

Then getting the sum of number of passengers in all trains by iterating through an array P and sum up all elements in O(m). Then substracting the sum with the number of spaces used we got from the max flow, this would leave the smallest number of passengers that need to be rescued due to unable to get to any stations because we substract by the maximum number of passengers from all trains that could get to any stations.

The graph has at most m + 2 vertices with 2m + mn edges requiring O(mn). Also, the maximum flow is bounded above by the capacity of the aforementioned cut, which is at most mn. Therefore, the Edmonds-Karp algorithm runs in O(E|f|) = O((mn)^2) time. Overall time complexity is O(m + mn + (mn)^2) = O((mn)^2) which is withnin time requirement.