Task Shopping

This problem uses a technique called fracturing search. It's very similar to using a modified Dijkstra's algorithm to find the K-th shortest path, but the problem with a naive implementation is that the number of states and transitions we need might be huge. We want an $O(1)$ transition and $O(K)$ states.

First let's assume that there is only one type of item. We sort all the items in non-decreasing order (of cost) and add the first x_0 of them to our base cost. We will call this the base state: the one that costs the least, with a cost of our "base cost" variable. Imagine that we have pointers (like arrows) pointing at each of the first x_0 items, indicating which ones we are taking. The way we will get to an arbitrary state is by first moving the rightmost pointer to the right, one step at a time until it reaches the rightmost item we need. We will never touch this pointer again. Let's say that we "glue it down". Now we begin moving the second rightmost pointer up until it reaches the second rightmost item we plan on taking and glue it down. Repeat this for every pointer we have. Notice that we never have two pointers point at the same item and we never "jump" a pointer over another one as it moves right. We may also add new pointers if we need to add more pointers to reach this arbitrary state. We will add additional pointers pointing to the leftmost item only when it is not already being pointed at.

We can let our state be $(j, cnt, maxr)$. j means that the rightmost pointer that hasn't been glued down is at index from the left. When $cnt < x_0$, cnt means how many pointers have been moved from their original spots. When $cnt \geq x_0$, cnt means how many pointers we currently have in our state. $maxr$ is the index of the leftmost pointer that has been glued down. This is needed so that we don't move future pointers at or beyond this point.

The only transitions we need are:

1. Moving j right by one.
2. Gluing down the current j and moving the next pointer (that is among one of the original x_0 ones) to the right by one.
3. Creating a new pointer that points to the first item.

Notice that every state corresponds to a unique set of choices. We have an $O(1)$ transition and we only need to visit exactly K states.

To extend our solution to work with multiple rows, we can add an i to our state, representing which row we're currently processing. We will process the rows from top to bottom and add transitions to go from the current state $(i, j, cnt, maxr)$ to a state in the following row. When we do this, we need to make sure that the new state represents a different set of pointers, otherwise our algorithm would not work. We can do this by forcing ourselves to, in addition to going to the next row, moving the rightmost pointer on the next row to the right by one. Let $base(i)$ denote the "base state" of row i, that which has the last cost. Let $right(i)$ denote the base state but with the rightmost pointer moved right by one. What if we want to skip a row? We can create a transition from the $right(i)$ to $right(i+1)$, but also subtracts the cost difference between $right(i)$ and $base(i)$. This essentially skips row i without moving any pointers in its row. To ensure that we visit our states in non-decreasing order of cost, we need to first order the rows accordingly. Watch out for numerous edge cases when implementing this.

Time Complexity: $O(N \log N + M \log M + K \log K)$.