First of all, I want to point out that my solution of previous assignment in which we had to implement a trie and **findItemSets** method does not cover all the cases. I mended the code just by moving the crucial if statement which checks if we reached a leaf node in the trie to the top of the method. Previously, this **if** was inside the loop and therefore it was impossible for it to reach last values specified in transactions. For example, given a transaction: 1, 3, 5. Our current trie contains following itemSets: [1, 3], [3, 5], [1, 5]. So, in my previous code, it didn't count [3, 5], because of for-loop's conditional check, i.e. after reaching last value 5 from transaction it goes deeper down through the recursive tree and was stopped by the for-loop's condition, because the **startindex** was bigger than the size of transaction, so the crucial **if** checking whether we reached a leaf node, couldn't be executed.

Secondly, I will start explaining **getSubSets** method. This is just simple combinatorics problem. The idea is to take values standing on indexes which has not yet been taken. Every time we take a new value, we mark the index as "taken", then iterate further by and add values standing on "untaken" indexes up to k-1 inclusively. The rest is self-explanatory, I guess.

Next, I will try to explain **getCandidateItemSets** method. Firstly, as said in the README file, when generating new candidates from previous results produced at passNum-1, each pair of previous results having the same subpart till the last element exclusive should be used as following to construct new candidate: firstly take the similar prefix and add the smallest last value to the end from the sequence forming the pair, then from the next sequence from pair, put its respective last value. Now, how do we do this, when having N sequences from previous pass each having **itemSetSize** equal to M. I did it like following:

1. Find hashes for prefixes lasting till the last element for each sequence.
2. Conduct self-joining (O(N^2)) by comparing their hashes (O(1)).
3. If their hashes are equal, produce new candidate as said in the text and add it to the candidates array.

Overall complexity is O(MN^2), but for most casual tests, it will be just fine. After that, I do pruning and if some **itemSet** is not appropriate, I delete it. That way, new candidates are produced.

The map method for passNum=1 is straightforward, for each encountered value in the transaction list, we emit it with value equal to 1.

The map method for passNum>1 is also pretty simple. But here, we start using trie which were contains all our generated candidates for current passNum. For each passed passed transaction to the **map** method, we call trie's **findItemSets** method, which gives us all the subsets from a transaction matching with trie's containing **itemSets**. Therefore, the matching ones are those candidates which we encountered in each transaction, we should simply emit each of them with value 1.

The reducer's task is to combine all similar keys and sum up over their representative values. We know that we have bunch of keys and each of their values is equal to 1. So, if we simply sum them and normalize our map to have only unique keys, then as a result we will have our output for the current passNum.

Now, answering on the question of why we do generate (k+1)-newItemSets only when two ItemSets have exactly similar prefixes up to k-1. We do so, because it allows us to reduce our candidates' space, that is removing unnecessary candidates. We know that candidates are produced from previous output results, also we know that the results are with support greater or equal to the **minSupport**, now self-joining everything will definitely produce unwanted duplicates, but doing as we do will preserve uniquely.

Lastly, regarding question of how much solution space is reduced by pruning. In my implementation, I think, pruning is done before the actual **prune** method, so the **prune**, just does more safety checking. We know, that if there are N sequences in the output from previous passNum, then when producing new candidates by self-joining, without pruning, we will have N*(N-1) candidates, whereas doing with pruning in the worst case only we will get the same number of candidates when all the prefixes of sequences are the same. But, because we know, that usually this is not the case, we can take a good advantage of it.