Performance of the two algorithms:

The first algorithm works early when almost all transactions contribute to the count.

The second algorithm works well whem only a small subset of the transactions contribute to the count (i.e. later in the algorithm)

The way to go is a hybrid algorithm:

The algorithm uses counts of candidates in Ck to estimate Ck'.

When Ck' gets below a threashold, the second algorithm is called.

Theoretical Analysis:

There is a trivial upper bound of

O(|c|.|D|).

Here |c| is the sum of sizes of all candidates
|D| is the size of the database.

Lower bounds and average case performance is more involved.

Generating Rulesets:

```
for all large k itemsets Lk (k >= 2){
        H1 = {consequents of rules from Lk with one item in the consequent}
        generate_rules(Lk, H1)
}

generate_rules(Lk: set of large k-itemsets, Hm: set of m-item consequents)
        if (k > m+1) {
                Hm+1 = generate all superset consequents (candidates) of Hm
                for all hm+1 in Hm+1 {
                        conf = support(Lk) / support (Lk - hm+1);
                        if (conf > minconf)
                                output Lk - hm+1 => hm+1
                        else
                                delete hm+1 from Hm+1
                }
                generate_rules(Lk, Hm+1)
}
```

Generating Rules:

For all large itemsets, we output all

a => l - a

such that

support (l) / support (a) > minconf

It is easy to see that

if AB => CD holds then

ABC => D and
ABD => C

hold as well.

Example:

Database		C1’		L1	
TID	items	TID	Set-of-Itemsets	Itemset	Support
100	1 3 4	100	{{1}, {3}, {4}}	{1}	2
200	2 3 5	200	{{2}, {3}, {4}}	{2}	3
300	1 2 3 4	300	{{1}, {2}, {3}, {5}}	{3}	3
400	2 5	400	{{2}, {5}}	{5}	3

C2		C2’		L2	
Itm.	Support	TID	Set of itemsets	Itemset	Support
{1,2}	1	100	{{1,3}}	{1,3}	2
{1,3}	2	200	{{2,3},{2,5},{3,5}}	{2,3}	2
{1,5}	1	300	{{1,2},{1,3},{1,5},	{2,5}	3
{2,3}	3		{2,3},{2,5},{3,5}}	{3,5}	2
{2,5}	3	400	{{2,5}}		
{3,5}	2				

C3		C3’		L3	
Itm.	Support	TID	Set of itemsets	Itemset	Support
{2,3,5}	2	200	{{2,3,5}}	{2,3,5}	2
		300	{{2,3,5}}		

Improving the Algorithm:

Basis: If a certain itemset if contained in a set of transactions, a superset of the itemset can only be contained in a subset of the set of transactions.

Therefore, instead of running through all the transactions on multiple passes, go through only the ones relevant to the itemset the candidate was generated from.

```
L1 = {large 1-itemsets}
C1' = database D
for (k = 2, lk-1 != 0; k++){
    Ck = generate candidates from Lk-1
    Ck' = 0
    for all entries t in C'k-1, do {
        Ct = {c in Ck such that c[1]... c[k-1] is in t or c[1] ... c[k-2]c[k] is in t}
        for all the candidates in Ck,
            c.count ++;
        if (Ct != 0) C'k += <t.TID, Ct>
    }
}
```

Subset functions:

Subset functions are generally implemented as a hash tree.

Leaf nodes of the tree store list of itemsets
Interior nodes store a hash function.

To populate the tree:

Start with an empty node and populate with subsets. When the number exceeds a bound, the node divides into two over a hash function.

Start at a node and make hash decisions on internal nodes.
When you reach the leaf, you have the subsets.

Generating Ck from Lk-1:

Basis of Algorithm:

Joining Lk-1 with Lk-1 to obtain a superset of the final set of candidates:

Say p and q are itemsets such that p and q belong to Lk-1.
Combine the two itemsets if they share the first k-2 items.
This is a candidate for being in Ck.

Delete all such combinations for which some k-1 subset of this combination is not in Lk-1.

For example

L3 = {{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {1, 3, 5}, {2, 3, 4}}

after join,

C4 = {{1, 2, 3, 4}, {1, 3, 4, 5}}

Now {1, 3, 4, 5} is pruned because {1, 4, 5} is not in Lk.

Algorithm Select large itemsets:

```
L1 = {Large 1-itemsets}
for (k = 2; Lk-1 != 0; k++)
{
        Ck = generate k itemset candidates from Lk-1;
        for all transactions t in D
        {
                Ct = subset (Ck, t)
                        // candidates in t
                C.count ++;
        }
        Lk = {c in Ck with count > minsup }
}
return the union of all Lks.
```

Algorithm Notation:

Assume that Transactions and Itemsets are sorted in lexicographic order.

The number of itemsets is the size.
An itemset of size k is called the k-itemset.

c[1], c[2], ..., c[k]: k itemset C

With each itemset, we have a count to store the support:

Lk: Set of large k itemsets (contains the itemset as well as the support).
Ck: Set of candidate k itemsets (potentially large itemsets).
C’k: Set of candidate k itemsets along with the transactions they appear in.

Computing Large Itemsets:

Basis of the algorithm:

Subsets of large itemsets are also large.

Conversely, large itemsets can be constructed by constructing supersets of smaller large itemsets.

The algorithm makes multiple passes over the transactions.

In each pass, the algorithm starts with a seed set of large itemsets (candidate itemsets). Of these, the ones with the minimum support are retained for the next pass and others are dropped.

The process continues until no further items are added.

There are two approaches to this:

- Compute from transactions
  (Although this avoids repeated passes over the datasets, it may require keeping a large number of counts for itemsets that may eventually be eliminated).

- Pass by pass
  (Prune as we go).

Association Rule Mining:

The problem can be decomposed into two parts:

1. Find all combinations of items that have the minimum support.
These combinations are also called large itemsets. All other itemsets are called small itemsets.

2. Use the large itemsets to generate desired rules:

Say ABCD and AB are large itemsets.

Then, AB => CD holds with a confidence that is given by:

r = support (ABCD) / support (AB)

Problem Specification:

An association rule is an implication of the form X => y

here X, and Y are both subsets of I and have nothing in common.

Therefore rules of the form

ABC => AD do not make sense.

The confidence of X => Y c is given by

c = Number of transactions that contain XY / Number of transactions that contain X.

The support for the hypothesis s is

s = Fraction of transactions with either X or Y appearing.

Association Rule Mining: Given a set of transactions D, the association problem is to generate all association rules that have a minimum support (minsup) and minimum confidence (minconf).

Formal Problem Statement:

Let:

$I = \{i_1, i_2, ..., i_m\}$ be a set of literals called items
$D = \{T_1, T_2, ..., T_n\}$ be a set of transactions

T_i has a transaction ID (TID) of i and is a set such that $T_i \subseteq I$

The database can be thought of as the following table:

	i_1	i_2	i_3	i_4	i_5
T_1	0	1	1	0	0
T_2	1	1	0	0	0
T_3	0	0	0	0	1

And so on.

Algorithms for Generating Associations:

Associations are rules of the form:

98% of the people who bought tires and accessories at Sears also had their cars serviced there. (From a business standpoint, this means: offer discounts on prices of tires and accessories and fleece the people on service consts).

Associations are detected from tables of the form

Person A = {Chips, Salsa, Cookies, Crackers, Coke, Beer}
Person B = {Lettuce, Spinach, Oranges, Celery, Apples, Grapes}
Person C = {Chips, Salsa, Frozen Pizza, Frozen Cake}
Person B = {Lettuce, Spinach, Milk, Butter}

We are trying to deduce from this that Chips and Salsa go together; and Lettuce and Spinach go together.

Infact, we would like to hypothesize that Chips => Salsa and Lettuce => Spinach.