Association Rules

(a) Define "support" and "confidence". (5 marks)

Given a set of transactions T, Association Rule Mining finds rules that will predict the occurrence of an item based on the occurrences of other items in the transaction. Such rules are implication expressions of the form X → Y, where X and Y are itemsets. In order to select interesting rules from the set of all possible rules, constraints on various measures of significance and interest are used. The best-known constraints are minimum thresholds on support and confidence. Support and Confidence are Rule Evaluation Metrics.

Support is an indication of how frequently the itemset appears in the dataset. The support of X → Y with respect to T is defined as the proportion of transactions t in the dataset which contains both the itemsets X and Y. It represents the coverage of the rule in T.

$$\mathrm{supp}(X) = \frac{|\{t \in T; X \subseteq t\}|}{|T|}$$

The argument of supp() is a set of preconditions, and thus becomes more restrictive as it grows (instead of more inclusive).

Confidence is an indication of how often the rule has been found to be true. The confidence value of a rule, X → Y , with respect to a set of transactions T, measures the proportion of the transactions that contains X which also contains Y. It represents the Accuracy of the rule.
Confidence is defined as:

$$\mathrm{conf}(X \Rightarrow Y) = \mathrm{supp}(X \cup Y)/\mathrm{supp}(X)$$

Example database with 5 transactions and 5 items

transaction ID	*milk*	*bread*	*butter*	*beer*	*diapers*
1	1	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	1	1	1	0	0
5	0	1	0	0	0

In the example dataset, the itemset X= {beer,diapers} has a support of 1/5=0.2 since it occurs in 20% of all transactions (1 out of 5 transactions).
The rule {butter,bread} → {milk} has a confidence of 0.2/0.2=1.0 in the database, which means that for 100% of the transactions containing butter and bread the rule is correct (100% of the times a customer buys butter and bread, milk is bought as well).

Bonus 😉 (You can thank me later if this is needed in the exam):
The lift of a rule is defined as the ratio of the observed support to that expected if X and Y were independent:

$$\mathrm{lift}(X \Rightarrow Y) = \frac{\mathrm{supp}(X \cup Y)}{\mathrm{supp}(X) \times \mathrm{supp}(Y)}$$

(b) <u>Define the notion of frequent itemset.</u>

A frequent itemset typically refers to a set of items that often appear together in a transactional data set—for example, milk and bread, which are frequently bought together in grocery stores by many customers.
A set of items is referred to as an itemset (*I*). An itemset that contains *k* items is a k-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. If the relative support of an itemset *I* satisfies a prespecified minimum support threshold (i.e., the absolute support of *I* satisfies the corresponding minimum support count threshold), then I is a frequent itemset. The set of frequent k-itemsets is commonly denoted by L_k.

If an itemset is frequent, each of its subsets is frequent as well. A long itemset will contain a combinatorial number of shorter, frequent sub-itemsets. For example, a frequent itemset of length 100, such as $\{a_1, a_2, :::, a_{100}\}$, contains

- $\binom{100}{1} = 100$ frequent 1-itemsets: $\{a_1\}, \{a_2\}, \ldots, \{a_{100}\}$
- $\binom{100}{2}$ frequent 2-itemsets: $\{a_1, a_2\}, \{a_1, a_2\}, \ldots, \{a_{99}, a_{100}\}$
- and so on

The total number of frequent itemsets that it contains is thus

$$\binom{100}{1} + \binom{100}{2} + \cdots + \binom{100}{100} = 2^{100} - 1 \approx 1.27 \times 10^{30}$$

This is too huge a number of itemsets for any computer to compute or store. To overcome this difficulty, we introduce the concepts of closed frequent itemset and maximal frequent itemset.

An itemset X is closed in a data set D if there exists no proper super-itemset Y such that Y has the same support count as X in D. An itemset X is a closed frequent itemset in set D if X is both closed and frequent in D. An itemset X is a maximal frequent itemset (or max-itemset) in a data set D if X is frequent, and there exists no super-itemset Y such that $X \subset Y$ and Y is frequent in D.

(c) <u>Define the notion of frequent itemset mining.</u>

Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining such patterns from their databases. The discovery of interesting correlation relationships among huge amounts of business transaction records can help in many business decision-making processes such as catalog design, cross-marketing, and customer shopping behavior analysis.

A typical example of frequent itemset mining is market basket analysis. This process analyzes customer buying habits by finding associations between the different items that customers place in their “shopping baskets”. The discovery of these associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers.

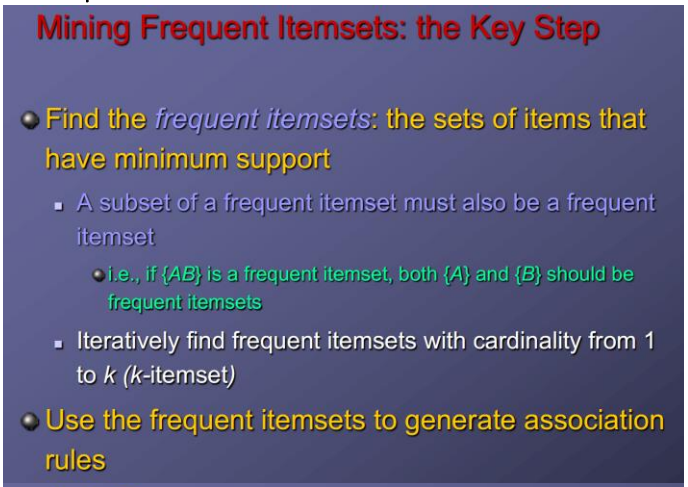
(d) <u>Given a set of large item-sets L, a minimum support s and minimum confidence α, write an algorithm that accepts as input L, s, and α, and outputs a set R of all association rules satisfying s and α. (9 marks)</u>

Definition of Apriori Algorithm

- The Apriori Algorithm is an influential algorithm for mining frequent item sets for Boolean association rules.
- Apriori uses a "bottom up" approach, where frequent subsets are extended one item at a time (a step known as candidate generation, and groups of candidates are tested against the data.

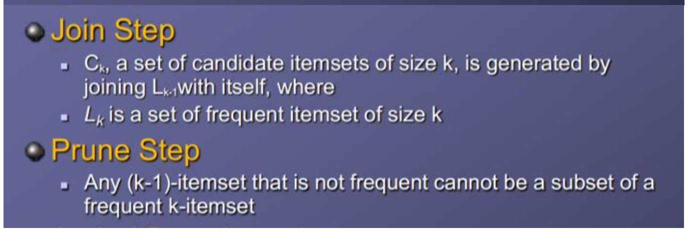
Below are the apriori algorithm steps:

1. Scan the transaction data base to get the support 'S' each 1-itemset, compare 'S' with min_sup, and get a support of 1-itemsets,
2. Use join to generate a set of candidate k-item set. Use apriori property to prune the unfrequented k-item sets from this set.



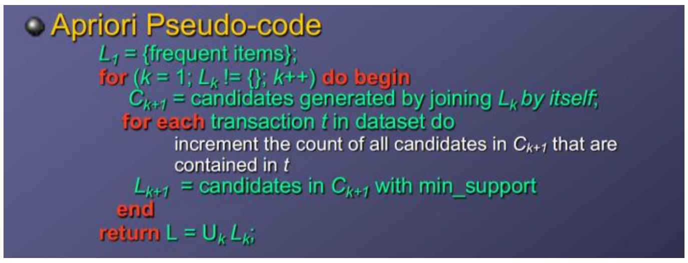
Mining Frequent Itemsets: the Key Step

- Find the *frequent itemsets*: the sets of items that have minimum support
  - A subset of a frequent itemset must also be a frequent itemset
    - i.e., if *{AB}* is a frequent itemset, both *{A}* and *{B}* should be frequent itemsets
  - Iteratively find frequent itemsets with cardinality from 1 to *k* (*k*-itemset)
- Use the frequent itemsets to generate association rules



- Join Step
  - C_k, a set of candidate itemsets of size k, is generated by joining L_{k-1} with itself, where
  - L_k is a set of frequent itemset of size k
- Prune Step
  - Any (k-1)-itemset that is not frequent cannot be a subset of a frequent k-itemset

3. Scan the transaction database to get the support 'S' of each candidate k-item set in the given set, compare 'S' with min_sup, and get a set of frequent k-item set
4. If the candidate set is NULL, for each frequent item set 1, generate all nonempty subsets of 1.



- Apriori Pseudo-code

```
L_1 = {frequent items};
for (k = 1; L_k != {}; k++) do begin
    C_{k+1} = candidates generated by joining L_k by itself;
    for each transaction t in dataset do
        increment the count of all candidates in C_{k+1} that are
        contained in t
    L_{k+1} = candidates in C_{k+1} with min_support
end
return L = U_k L_k;
```

5. For every nonempty subsets of 1,
   output the rule "s → (1-s)" if confidence C of the rule "s → (1-s)" > min_conf
   (Add step 5 to the pseudocode in the image above for the rule output)
6. If the candidate set is not NULL, go to step 2.

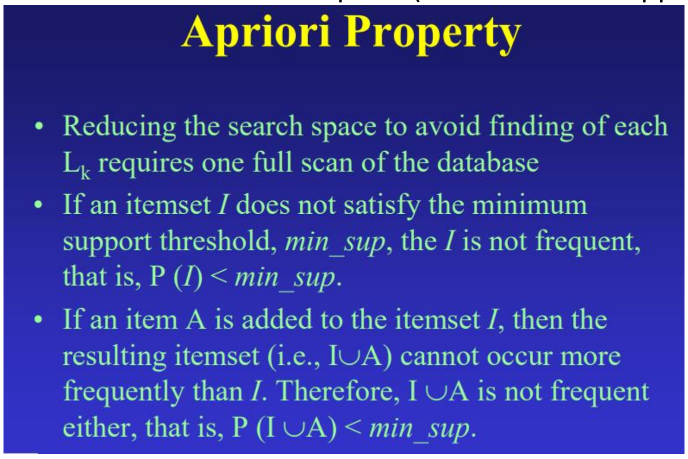
(e) <u>To reduce the search space when looking for frequent itemsets, an important property called Apriori Property is used. What is the Apriori property? (5 marks)</u>

Apriori property is used to reduce the search space, and is defined as: All nonempty subset of frequent items must be also frequent. It holds due to the following property of the support measure

$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

It is anti-monotone, in the sense that if a set cannot pass a test, all its super sets will fail the same test as well, i.e., if s(X) < minsup, then s(Y) <= s(X) < minsup

This means, at each level k, we have k-item sets which are frequent (have sufficent support). At the next level, the k+1-item sets we need to consider must have the property that each of their subsets must be frequent (have sufficent support).



Apriori Property

- Reducing the search space to avoid finding of each L_k requires one full scan of the database
- If an itemset *I* does not satisfy the minimum support threshold, *min_sup*, the *I* is not frequent, that is, P (*I*) < *min_sup*.
- If an item A is added to the itemset *I*, then the resulting itemset (i.e., I∪A) cannot occur more frequently than *I*. Therefore, I ∪A is not frequent either, that is, P (I ∪A) < *min_sup*.

(f) <u>The Apriori algorithm has two main steps: The join step and the prune step. Define these join and prune steps. (5 marks)</u>

"How is the Apriori property used in the algorithm?" To understand this, let us look at how L_{k-1} is used to find L_k for $k \geq 2$. A two-step process is followed, consisting of **join** and **prune** actions.

1. **The join step**: To find L_k, a set of **candidate** k-itemsets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k. Let l_1 and l_2 be itemsets in L_{k-1}. The notation $l_i[j]$ refers to the jth item in l_i (e.g., $l_1[k-2]$ refers to the second to the last item in l_1). For efficient implementation, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the $(k-1)$-itemset, l_i, this means that the items are sorted such that $l_i[1] < l_i[2] < \cdots < l_i[k-1]$. The join, $L_{k-1} \bowtie L_{k-1}$, is performed, where members of L_{k-1} are joinable if their first $(k-2)$ items are in common. That is, members l_1 and l_2 of L_{k-1} are joined if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \cdots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$. The condition $l_1[k-1] < l_2[k-1]$ simply ensures that no duplicates are generated. The resulting itemset formed by joining l_1 and l_2 is $\{l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]\}$.
2. **The prune step**: C_k is a superset of L_k, that is, its members may or may not be frequent, but all of the frequent k-itemsets are included in C_k. A database scan to determine the count of each candidate in C_k would result in the determination of L_k (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to L_k). C_k, however, can be huge, and so this could involve heavy computation. To reduce the size of C_k, the Apriori property is used as follows. Any $(k-1)$-itemset that is not frequent cannot be a subset of a frequent k-itemset. Hence, if any $(k-1)$-subset of a candidate k-itemset is not in L_{k-1}, then the candidate cannot be frequent either and so can be removed from C_k. This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

(g) Find all frequent items for the above database using the Apriori algorithm. (5 marks)

Solve problem using the following algorithm

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- *min_sup*, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

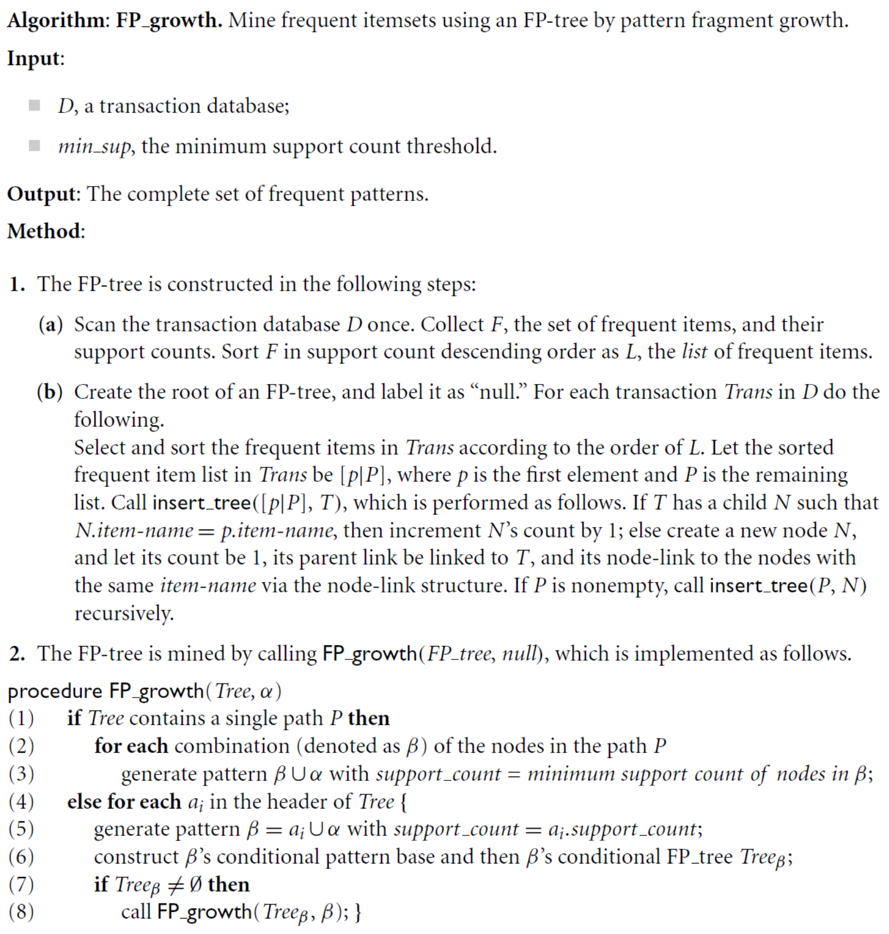
```
(1)   $L_1$ = find_frequent_1-itemsets(D);
(2)   for ($k = 2; L_{k-1} \neq \phi; k{+}{+}$) {
(3)      $C_k$ = apriori_gen($L_{k-1}$);
(4)      for each transaction $t \in D$ { // scan $D$ for counts
(5)         $C_t$ = subset($C_k$, $t$); // get the subsets of $t$ that are candidates
(6)         for each candidate $c \in C_t$
(7)            c.count++;
(8)      }
(9)      $L_k = \{c \in C_k | c.count \geq min\_sup\}$
(10)  }
(11)  return $L = \cup_k L_k$;

procedure apriori_gen($L_{k-1}$:frequent $(k-1)$-itemsets)
(1)   for each itemset $l_1 \in L_{k-1}$
(2)      for each itemset $l_2 \in L_{k-1}$
(3)         if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2])$
               $\wedge ... \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$ then {
(4)            $c = l_1 \bowtie l_2$; // join step: generate candidates
(5)            if has_infrequent_subset($c$, $L_{k-1}$) then
(6)               delete $c$; // prune step: remove unfruitful candidate
(7)            else add $c$ to $C_k$;
(8)         }
(9)   return $C_k$;

procedure has_infrequent_subset($c$: candidate $k$-itemset;
         $L_{k-1}$: frequent $(k-1)$-itemsets); // use prior knowledge
(1)   for each $(k-1)$-subset $s$ of $c$
(2)      if $s \notin L_{k-1}$ then
(3)         return TRUE;
(4)   return FALSE;
```

(h) <u>Find all frequent items using the FP-growth algorithm. (5 marks)</u>

Solve problem using the following algorithm:



Algorithm: FP_growth. Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- *D*, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:
   - **(a)** Scan the transaction database *D* once. Collect *F*, the set of frequent items, and their support counts. Sort *F* in support count descending order as *L*, the *list* of frequent items.
   - **(b)** Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in *D* do the following.
     Select and sort the frequent items in *Trans* according to the order of *L*. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, $\alpha$)
(1)   if Tree contains a single path P then
(2)       for each combination (denoted as $\beta$) of the nodes in the path P
(3)           generate pattern $\beta \cup \alpha$ with support_count = minimum support count of nodes in $\beta$;
(4)   else for each $a_i$ in the header of Tree {
(5)       generate pattern $\beta = a_i \cup \alpha$ with support_count = $a_i$.support_count;
(6)       construct $\beta$'s conditional pattern base and then $\beta$'s conditional FP_tree $Tree_\beta$;
(7)       if $Tree_\beta \neq \emptyset$ then
(8)           call FP_growth($Tree_\beta$, $\beta$); }
```

(i) <u>Compare the efficiency of the two frequent itemset mining processes. (5 marks)</u>

The Apriori algorithm is based on the fact that if a subset S appears k times, any other subset S' that contains S will appear k times or less. So, if S doesn't pass the minimum support threshold, neither does S'. There is no need to calculate S', it is discarded a priori.

Disadvantages of Apriori

- It needs to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
- It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets. To compute those with support more than min_support, the database needs to be scanned at every level. It needs (n +1) scans, where n is the length of the longest pattern.

FP-Growth

Frequent Pattern growth, or simply FP-growth, adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each

database separately. For each "pattern fragment," only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the "growth" of patterns being examined.
Advantages of FP Growth

- The biggest advantage found in FP-Growth is the fact that the algorithm only needs to scan the transaction database twice, as opposed to apriori which scans it once for every iteration.
- Another huge advantage is that it removes the need to calculate the pairs to be counted, which is very processing heavy, because it uses the FP-Treee. This makes it O(n) which is much faster than apriori.
- The FP-Growth algorithm stores in memory a compact version of the database.

While the Apriori algorithm can find all frequent itemsets within the initial dataset, it suffers from two non-trivial costs.

(j) What are these two non-trivial costs? (5 marks)

1. It needs to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
2. It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets. To compute those with support more than min_support, the database needs to be scanned at every level. It needs (n +1) scans, where n is the length of the longest pattern.

(k) Explain how the FP-growth method avoids the two costly problems of the Apriori algorithm. (5 marks)

The 2 costly problems of Apriori algorithm are:

1. It needs to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
2. It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets. To compute those with support more than min_support, the database needs to be scanned at every level. It needs (n +1) scans, where n is the length of the longest pattern.

FP-growth method avoids the two costly problems because

1. The FP-growth method transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs.
2. The biggest advantage found in FP-Growth is the fact that the algorithm only needs to scan the transaction database twice, as opposed to apriori which scans it once for every iteration.
3. When the database is large, it is sometimes unrealistic to construct a main memory based FP-tree. An interesting alternative is to first partition the database into a set of projected databases, and then construct an FP-tree and mine it in each projected

database. This process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory.
4. A study of the FP-growth method performance shows that it is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm.

(l) <u>One of the benefits of the FP-tree structure is Compactness. Explain why FP-growth method is compact. (5 marks)</u>

FP Growth algorithm compresses a large database into a compact, Frequent-Pattern tree (FP-tree) structure, which is highly condensed, but complete for frequent pattern mining. It avoids costly database scans as there is no candidate generation, because it uses sub-database only. Therefore, the algorithm only needs to scan the transaction database twice.

The following reasons make the FP-growth method very compact.
1. Reduce irrelevant information—infrequent items are gone
2. Frequency descending ordering: more frequent items are more likely to be shared
3. Never be larger than the original database (if not count node-links and counts)

(m) <u>Give the necessary steps of the FP-growth algorithm.</u>

The process for FP Tree construction is simple. An FP-tree is constructed from a Transaction DB

a. Scan Dataset D to find frequent 1-itemsets (single item patterns)
b. Order frequent items in frequency descending order
c. Scan Dataset D again to construct FP-tree

Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- *D*, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:

(a) Scan the transaction database *D* once. Collect *F*, the set of frequent items, and their support counts. Sort *F* in support count descending order as *L*, the *list* of frequent items.

(b) Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in *D* do the following.
Select and sort the frequent items in *Trans* according to the order of *L*. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that *N.item-name* = *p.item-name*, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.

2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, $\alpha$)
(1)  if Tree contains a single path P then
(2)     for each combination (denoted as $\beta$) of the nodes in the path P
(3)        generate pattern $\beta \cup \alpha$ with support_count = minimum support count of nodes in $\beta$;
(4)  else for each $a_i$ in the header of Tree {
(5)     generate pattern $\beta = a_i \cup \alpha$ with support_count = $a_i$.support_count;
(6)     construct $\beta$'s conditional pattern base and then $\beta$'s conditional FP_tree $Tree_\beta$;
(7)     if $Tree_\beta \neq \emptyset$ then
(8)        call FP_growth($Tree_\beta$, $\beta$); }
```

(n) <u>The FP-growth algorithm adopts the strategy of divide-and-conquer. What is the main advantage of this strategy over the one used in the Apriori algorithm?</u>

Frequent Pattern growth, or simply FP-growth, adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each database separately. For each "pattern fragment," only its associated data sets need to be examined.

The Main Advantage:

This approach may substantially reduce the size of the data sets to be searched, along with the "growth" of patterns being examined, in comparison to Apriori Algorithm which has the following non-trivial costs associated with it:

1) It needs to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
2) It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets. To compute those with support more than min_support, the database needs to be scanned at every level. It needs (n +1) scans, where n is the length of the longest pattern.