The Apriori Algorithm for Finding Association Rules

```
function apriori (I, T, s_min, c_min, k_max)        (* apriori algorithm for association rules *)
begin
   k  := 1;                                         (* — find frequent item sets *)
   C_k := ⋃_{i∈I} {i};                              (* start with single element sets *)
   F_k := prune(C_k, T, s_min);                     (* and determine the frequent ones *)
   while F_k ≠ ∅ and k ≤ k_max do begin             (* while there are frequent item sets *)
      C_{k+1} := candidates(F_k);                   (* create item sets with one item more *)
      F_{k+1} := prune(C_{k+1}, T, s_min);          (* and determine the frequent ones *)
      k       := k + 1;                             (* increment the item counter *)
   end;
   R := ∅;                                          (* — generate association rules *)
   forall f ∈ ⋃_{j=2}^{k} F_j do begin              (* traverse the frequent item sets *)
      m  := 1;                                      (* start with rule heads (consequents) *)
      H_m := ⋃_{i∈f} {i};                           (* that contain only one item *)
      repeat                                        (* traverse rule heads of increasing size *)
         forall h ∈ H_m do                          (* traverse the possible rule heads *)
            if s(f)/s(f−h) ≥ c_min                  (* if the confidence of the rule *)
            then R   := R ∪ {[(f − h) → h]};        (* is high enough, add it to the result, *)
            else H_m := H_m − {h};                  (* otherwise discard the rule head *)
         H_{m+1} := candidates(H_m);                (* create rule heads with one item more *)
         m       := m + 1;                          (* increment the head item counter *)
      until H_m = ∅ or m ≥ |f|;                     (* until there are no more rule heads *)
   end;                                             (* or the antecedent would become empty *)
   return R;                                        (* return the rules found *)
end (* apriori *)

function candidates (F_k)                           (* generate candidates with k + 1 items *)
begin
   C := ∅;                                          (* initialize the set of candidates *)
   forall f_1, f_2 ∈ F_k                            (* traverse all pairs of frequent item sets *)
   with  f_1 = {i_1, ..., i_{k−1}, i_k}             (* that differ only in one item and *)
   and   f_2 = {i_1, ..., i_{k−1}, i'_k}            (* are in a lexicographic order *)
   and   i_k < i'_k do begin                        (* (the order is arbitrary, but fixed) *)
      f := f_1 ∪ f_2 = {i_1, ..., i_{k−1}, i_k, i'_k};  (* the union of these sets has k + 1 items *)
      if ∀i ∈ f : f − {i} ∈ F_k                     (* only if all k element subsets are frequent, *)
      then C := C ∪ {f};                            (* add the new item set to the candidates *)
   end;                                             (* (otherwise it cannot be frequent) *)
   return C;                                        (* return the generated candidates *)
end (* candidates *)

function prune (C, T, s_min)                        (* prune infrequent candidates *)
begin
   forall c ∈ C do                                  (* initialize the support counters *)
      s(c) := 0;                                    (* of all candidates to be checked *)
   forall t ∈ T do                                  (* traverse the transactions *)
      forall c ∈ C do                               (* traverse the candidates *)
         if c ∈ t                                   (* if the transaction contains the candidate, *)
         then s(c) := s(c) + 1;                     (* increment the support counter *)
   F := ∅;                                          (* initialize the set of frequent candidates *)
   forall c ∈ C do                                  (* traverse the candidates *)
      if s(c) ≥ s_min                               (* if a candidate is frequent, *)
      then F := F ∪ {c};                            (* add it to the set of frequent candidates *)
   return F;                                        (* return the pruned set of candidates *)
end (* prune *)
```