1. **factor_product:** Using the 2 cardinalities of given factors A&B we create a probability table by multiplying values for each possible combination within their cartesian product. The resulting probability table, encompassing the union of all random variables from factors A and B, along with their respective probability products, is referred to as the "factor_product"
2. **factor_marginalize:** The variables to be retained in the output factor after marginalization are computed first. Then their corresponding indices are found in the input factor's value table. Rows in the input factor table are grouped by the combination of output factors, and the sums across these combinations are calculated. This information is stored in a dictionary, mapping output variable combinations to their respective final probability values for constructing the output factor.
3. **observe_evidence:** For each factor in the input list, determine if it needs processing by evaluating intersections with the provided evidence variables.Only process factors with non-zero intersections. To process, identify the indices of each evidence variable based on their occurrence in the input factor table rows. During row iteration, check if evidence factor values match those in the row. If not, set the final probability for that row to 0.
4. **compute_joint_distribution:** To compute the joint distribution, the factor_product implementation is reused, and this operation is applied cumulatively to each of the factors in the provided factor list. The final factor_product result corresponds to the joint distribution across all the input factors provided.
5. **compute_marginals_naive:** To compute the marginal probability value of P(V|Evidence), the process involves initially computing the total joint probability across all factors to derive P(f1, f2, ... fn). Then, the observe_evidence function is applied to the provided evidence, resulting in P(f1, f2, ... fn | Evidence). Finally, the factor_marginalize operation is utilized on this distribution, marginalizing all variables except for V, thus yielding the required marginal, P(V|Evidence).
6. **compute_marginals_bp:** This process essentially mirrors the sum-product algorithm for computing all possible marginals P(Vi|E), with the advantage of being more time-efficient due to a single graph traversal in both directions. My implementation comprises several phases:
   In *Phase 0*, preprocessing identifies the root and establishes bidirectional parent-child relationships for all graph nodes.
   In *Phase 1*, during the upward pass, my implementation computes messages from its children using factor_product, incorporating the binary potential of its edge. During message passing, variables of the sender nodes are marginalized cumulatively, and computed mij values are stored in the messages array for later retrieval. This phase stops with the root since the root lacks a parent.
   In *Phase 2*, the downward pass begins by considering the root's unary potential and any previously received messages. Similar to previous message computations, this phase adheres to the message-passing protocol, collecting and forwarding messages while storing computed values in the messages array.
   In *Phase 3*, for each of the inference variables requiring marginal probabilities, my implementation considers all received messages from neighbors. Factor_product is applied cumulatively to these messages, followed by normalization to obtain the final marginal probabilities.
7. **factor_sum:** This implementation is akin to the factor_product implementation, with the key distinction being the utilization of "log space." In this context, instead of products, we perform summations by adding the individual values for each combined possibility in the cartesian product of both factors.
8. **factor_max_marginalize:** This implementation closely parallels the factor_marginalize implementation, with the key distinction being the comprehensive tracking of the state of each variable being marginalized. In this approach, a dictionary is maintained, mapping combinations of variables present only in the output factor to lists containing both the probability value in that row and the combination of variables to be marginalized along with their respective states. This dual storage of probability values and the variables to be marginalized simplifies the determination of maximum and argmax for a given combination of output random variables, which are then computed and stored in the resulting output factor.
9. **map_eliminate:** This approach, akin to compute_bp, operates as a max-sum algorithm in log space using the max operator, focusing on MAP and Maximal configuration. It comprises several phases. In *Phase 0*, evidence is observed, factors are transformed into log space, and preprocessing identifies the root and establishes bidirectional parent-child relationships in the graph.*Phase 1* involves replacing factor_product and factor_marginalize with factor_sum and factor_max_marginalize for the upward pass. In *Phase 2*, there's no need for a downward pass as the MAP is computable after the upward pass. The final factor_sum at the root, combined with max_marginalization, provides the log of the maximal probability and the best value for the root in the maximal configuration.*Phase 3* consists of recursively traversing the tree using the root's maximal configuration. By passing the maximizing value from parent to child and using the message between them, maximizing values for other variables in the graph are efficiently computed.