Mapping

```
mapping( base: List[string], target: List[string] ) -> List[str]:
  --------------------------------
  // assuming len(base) == n, len(target) == m
  // there are ((n choose 2) * (m choose 2) * 2) pairs
  possible_pairs = get_all_possible_pairs(base, target)

  // here we going to store the entities that already mapped.
  // the value in index i in both lists will be the map between them.
  // it is clear that both must be in the same length.
  base_already_map, target_already_map = [], []

  while len(base_already_map) < min(len(base), len(target)):
    // updating the possible pairs according to the entities that already mapped
    // the idea is to not break the entities that already mapped.
    update_possible_pairs(possible_pairs, base_already_map, target_already_map)

    // we want the pair with the best score.
    // the meaning of pair is for example: earth→electrons AND sun→nucleus.
    res = get_best_pair_mapping(possible_pairs)

    if res["score"] > 0:
      // updating the already mapped lists.
      // res["base"][0] → res["target"][0], res["base"][1] → res["target"][1]
      update_list(base_already_map, res["base"])
      update_list(target_already_map, res["target"])
    else:
      // no map found at all.
      break
  --------------------------------
  return [f"{b} → {t}" for b, t in zip(base_already_map, target_already_map)]
```

```
get_best_pair_mapping( pairs: List[List[Tuple[string]]] ) -> List[Tuple[string]]:
  mapping = []
  --------------------------------
  for pair in pairs:
    // pair is something like: [(earth, sun), (electrons, nucleus)]
    base_edge, target_edge = pair
    mapping.append(pair, get_score(base_edge, target_edge))
  --------------------------------
  return sorted(mapping, key=lambda x: [1], reverse=True)[0]
```

Clustering + score

```
get_score( e1: tuple, e2: tuple):
  --------------------------------
  score = 0
  // we count both directions, for example for mapping earth→electrons, sun→nucleus
  // we will count (earth:sun,electrons:nucleus) and (sun:earth,nucleus:electrons)
  for i in range(2):  // direction
    // e1 and e2 will flip in the second iteration (direction..)
    props1 = get_edge_props( e1 )  // List[str]
    props2 = get_edge_props( e2 )  // List[str]

    // this will create a full bipartite graph between props1 and props2
    similarity_edges = get_edges_weights( props1, props2 ) // List[Tuple[str, str, float]]

    // clustering is using AgglomerativeClustering of sklearn.cluster
    // https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html
    // distance_threshold  → how close the props in the cluster
    clusters_props1 = clustering( props1, distance_threshold ) // Dict[int, List[str]]
    clusters_props2 = clustering( props2, distance_threshold ) // Dict[int, List[str]]

    // between every two clusters (from the opposite side of the bipartite) we will take
    // only one edge, which will be the one with the maximum weight.
    clusters_edges = get_clusters_edges( similarity_edges, clusters_props1, clusters_props1 )

    // we want the maximum-weight of full bipartite matching
    // we will use networkx algorithm of minimum_weight_full_matching
    // https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.minimum_weight_full_matching.html
    best_matching = maximum_weight_full_matching( clusters_edges )
    score += sum([edge[2] for edge in best_matching])
  --------------------------------
  return score
```

```
get_edge_props( subject: string, object: string ) -> List[str]:
  --------------------------------
  props1 = get_props_from_quasimodo( subject, object, n_largest = 10 )  // sorted by plausibility
  props2 = get_props_from_google( subject, object )  // why|how do|does|did
  props3 = get_props_from_conceptnet( subject, object )  // sorted by concept-net weights
  --------------------------------
  return props1 + props2 + props3
```

```
get_edges_weights( props_edge_1: List[string], props_edge_2: List[string] ):
  --------------------------------
  edges = []
  for p1 in props_edge_1:
    for p2 in props_edge_2:
      // similarity is calculated by cosine-similarity.
      // https://pytorch.org/docs/stable/generated/torch.nn.CosineSimilarity.html
      edges.append( (p1, p2, similarity( p1,p2 )) )
  --------------------------------
  return edges
```