```
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( 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]
```