**Lina Ghanim**

My approach:

I converted the nuts into a representation of lists. Each nut is converted into a list that start with 1 and sequence counterclockwise.

For example:



This nut converts to (1 4 6 2 3 5)

If we place all the nuts this way then we will have a list of lists that represent the 7 nuts.

If we want to find the solution to the game the lists have to be ordered in a way where the first nut is the middle nut and for each other nut; (the first number = corresponding number of the middle nut) and (the second number = last number of previous nut).

Question 2:

I selected a specific nut game, permuted all the possibilities of ordering the nuts. For each permutation, I checked if the previous test that I mentioned applied to it. If not, I would try a different permutation.

Question 3:

I select a middle nut, try placing one nut at a time, check if the test passes on it. If yes, then I try placing another nut. If not, I go back and try another nut. If I rotate all the remaining nuts and don’t find a solution, I return nil.

My results:

I tested the \*nuts \*list in the Main method:

Question 2: (solve \*nuts\*)

Result: ((1 6 2 4 5 3) (1 4 6 2 3 5) (1 6 5 3 2 4) (1 4 3 6 5 2) (1 2 3 4 5 6) (1 6 4 2 5 3) (1 6 5 4 3 2))

Question 3: (dfs \*nuts\* 0)

Result: ((1 6 2 4 5 3) (1 4 6 2 3 5) (6 5 3 2 4 1) (2 1 4 3 6 5) (4 5 6 1 2 3) (5 3 1 6 4 2) (3 2 1 6 5 4))

Question 4: (cnt1 \*nuts\* 0)

Result: 1

Question 5: (graph 0 0 0 0)

Result: (789 211 0) 🡪 789 of the puzzles have 0 solutions, 211 have 1 solution, 0 have 2 solutions

Comment: The method –generate- does not work for puzzles greater than 9. The compiler prints a message: Heap exhausted during garbage collection: 0 bytes available, 16 requested. Because my generate method permutes all possible permutations. If I create a puzzle of n pieces randomly without using permute, it will work.