## 1b. English-language explanation (15 pts)

The aim of the pseudocode above is to implement the function fastest\_tour\_bf, which finds the shortest tour to traverse all lighthouses in an array L starting from a lighthouse start\_light. The function is implemented recursively using a brute-force approach, which evaluates all permutations of tours of L, to find the shortest route. At the start of the algorithm the start\_light is removed from L. The base case of the recursion if when no lighthouses remain in L after removing the start\_light, in which case the shortest tour is just the starting lighthouse and the tour time is 0.0. When one or more lighthouses remains, each “neighbor” lighthouse second\_light in L is recursively searched with second\_light as the new start\_light. For each recursive subtour starting at second\_light, the tour time is computed as the sum of the recursively computed best subtour of L starting at second\_light and the travel time between the starting lighthouse and second\_light. If this sum is less than the best\_time for all neighbors, it is assigned to best\_time and the subtour is assigned to best\_tour. The fastest subtour for all neighbors, best\_tour, is prepended with the starting node and returned. In this way all the fastest subtours starting at all neighbors of start\_light are compared, and the shortest is selected. An external function get\_travel\_time(start\_light,second\_light) is used to lookup the distance between two lighthouses

## 2a. Asymptotic bounds (15 pts)

Since all permutations of L, except the first lighthouse, are evaluated, the runtime should be Θ ((n-1)!), where n is the number of lighthouses in L. This can also be proved from the implementation. For each function call we perform several constant time operations, such as comparison, addition, hashtable lookup and get/set from start of list (we can use linked list to make this constant time) and n-1 recursive calls. This can be written in terms of the recurrence relation:

Expanding the recurrence:

Solving the recurrence and dropping the constant time terms:

Thus, we see the runtime of the algorithm is Θ ((n-1)!) for L with n node. I This analysis assumes that list\_minus is implemented in constant time (I am unsure about the set semantics in Python). If instead a naive linear time list function was used the runtime of Θ((n-1)!) \* O(n).

## Retrospection (10 pts)

The function fastest\_tour\_bf(start\_light, L) implements the pseudocode algorithm described above. The function finds the shortest tour of lighthouses in L starting from start\_light. In addition to computing the shortest tour, the number of recursive steps is tallied to estimate the runtime of the algorithm. The graph in 2b, verifies out runtime analysis. In fact, the number of steps in our algorithm is exactly (n)!, since our wrapper function generates L of length n excluding the starting light. This off-by-one error confused me during the initial implementation but debugging the code helped elucidate the difference in size between L and total lighthouses.

A few modifications were made to improve code quality. Strong type definitions were assigned to the parameters, return types and variables of the function and method documentation was provided. An alternative implementation of TRAVEL\_TIME() was implemented as get\_travel\_time(), which avoided variable shadowing between the function TRAVEL\_TIME and the map TRAVEL\_TIME. The function get\_travel\_time() also raises an expectation if the keys are not found in the map. Lastly a wrapper function was made to call fastest\_tour\_bf(start\_light, L) starting at each lighthouse in L. This allowed for easier testing. Test cases were defined and used to validate the implementation, ensuring code changes did not affect the efficacy of the algorithm. If possible, explicitly passing the TRAVEL\_TIME map to fastest\_tour\_bf() would have been preferred over global variable in get\_travel\_time().

Explain your results here. Your graph should "tell a story"... write the story in plain English here.

If you had a problem with your code, or graph, write down here how you followed the 1-2-3 Rule given in the Getting Started module. You should explain what you tried to do, what did/did not work, and what you would do next if you had additional time. \*\*Important: don't just turn in broken code, or graphs that don't match your analytic results!\*\* It's important to show that, if there was some problem, you recognized that there was an issue and what you were doing to resolve the disconnect. Bugs and mistakes do happen, time does run out, but you need to show what you understand of the problem in order to get at least partial credit.

The function fastest\_tour\_bf(start\_light, L) implements the pseudocode algorithm described above. The function finds the shortest tour of lighthouses in L starting from start\_light. In addition to computing the shortest tour, the number of recursive steps is tallied to estimate the runtime of the algorithm. The graph in 2b, verifies out runtime analysis. In fact, the number of steps in our algorithm is Θ(n-1)! - the red line, not Θ(n!) - the green line, where n is the total lighthouses. Since our algorithm searches through all permutations of L with start\_light in the first position, which is equivalent to all permutations of n-1. This off-by-one error confused me during the initial implementation but debugging the code helped elucidate the difference in size between L and total lighthouses.

A few modifications were made to improve code quality. Type definitions were assigned to the parameters, return types and variables of the function and method documentation was provided. An alternative implementation of TRAVEL\_TIME() was implemented as get\_travel\_time(), which raises an expectation if the keys are not found in the map. Lastly a wrapper function was made to call fastest\_tour\_bf(start\_light, L) starting at each lighthouse in L. This allowed for easier testing. Test cases were defined and used to validate the implementation, ensuring code changes did not affect the efficacy of the algorithm. If possible, explicitly passing the TRAVEL\_TIME map to fastest\_tour\_bf() would have been preferred over global variable in get\_travel\_time().