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

The aim of this code block is to implement the function fastest\_tour that finds the shortest path to traverse all nodes in an array L starting from a node x. The function is implemented recursively using a brute-force approach, which evaluates all permutations of tours of L, to find the shortest route. The base case of the recusion if when L contains a single node y, and the shortest tour from the starting node x is [x,y]. When L contains more than one node, each “neighbor” node y, in L is removed from L to form L\` and then recursively searched starting from y. The fastest subtour for all neighbors is prepended with the starting node and returned. Each pair of nodes is assigned a travel time, which can be queried using the get\_travel\_time(x,y) function. The fastest subtour and associated time is stored in two variables, min\_time and min\_tour, which are initially set to infinity and NULL, respectively. For each recursive subtour starting at node y, the tour time is computed as the sum of the best subtour of L starting at y and the travel time between the starting node and y. If this sum is less than the min\_time variable it is assigned to min\_time and the subtour is assigned to best tour. In this was all the shortest subtour starting at all neighbors of x are compared.

## 2a. Asymptotic bounds (15 pts)

Since all permutations of L are evaluated, the runtime should be O((n)!), where n is the number of nodes 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:

T(0) = C

Solving the recurrence and dropping the constant time terms:

Thus, we see the runtime of the algorithm is O((n)!) for L with n node. The starting node is not included in the count for L. If it was the runtime could be described by O((n-1))!. This 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 the algorithm would be O(n!) \* O(n) = O(n\*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.