**English-language explanation**

The function ‘fastest\_tour\_memo(start\_light, L)’ defines an algorithm to find the fastest tour through an array of lighthouses L starting from a lighthouse ‘start\_light’ using memoization. An external function, ‘get\_travel\_time(x,y)’ is used to lookup the travel time between two lighthouses. ‘fastest\_tour\_memo()’ begins by initializing an array ‘best\_tour’ to hold the lighthouse tour order. A variable ‘best\_time’ is used to keep track of the total travel time between the lighthouses in the tour and is initialized to infinity. The variable ‘all\_steps’, which is initialized to zero, counts the number of “computational steps” made by the program, as a way of estimating the Big-O runtime of the algorithm. Next the starting lighthouse is removed from L using an externally defined function ‘list\_minus()’. We now enter the core of the algorithm, the base case, memo case and recursive case. We use the technique of memoization to optimize the search for fastest tour. Memoization is the use of an external data structure to store previously computed results to subproblems to avoid performing duplicate work. In ‘fastest\_tour\_memo ()’ a global dictionary, ‘memo’ is defined and reset by calling code. We define the ‘memo\_lookup\_key ‘ to be a tuple with the starting lighthouse in the first position and a frozenset of the remaining lighthouses in the second position. As we will see later, we need to key by the starting lighthouse since the ‘best\_tour’ and ‘best\_time’ calculations take the starting light into account. The list of remaining lighthouses must be cast to a frozenset, since Python requires dictionary keys to be immutable to define a valid hash function. Frozensets also provide an advantage to tuples, as their equality test is order independent.

We can now describe the three cases. The memo hit case occurs if the ‘memo\_lookup\_key’ is present in ‘memo’. If so, we have a memo hit, which means the solution the best tour has already been computed. We can retrieve the ‘best\_tour’ and ‘best\_time’ from memo using the ‘memo\_lookup\_key’. We increment the global ‘memo\_hit’ counter and ‘all\_steps’ before returning ‘best\_tour’, ‘best\_time’ and ‘all\_steps’.

If ‘memo\_lookup\_key’ is not present in ‘memo’ and no remaining lighthouses remain, we have hit the base case. We precede by inserting the starting lighthouse into ‘best\_tour’, setting ‘best\_time’ to zero and adding an array [‘best\_tour’, ‘best\_time’] to ‘memo’, using the ‘memo\_lookup\_key’ defined above. We also increment ‘all\_steps’ to denote the constant time worked done by the base case. We then return ‘best\_tour’, ‘best\_time’ and ‘all\_steps’.

If there are one or more remaining lighthouses and the ‘memo\_lookup\_key’ is not in ‘memo’ we find the subproblem solution via recursion. We perform a for-loop over each remaining lighthouse to recursively calculate the best tour/time. For each ‘second\_light’ in the remaining lighthouses we recursively call ‘fastest\_tour\_memo()’ with ‘second\_light’ and ‘L’ as arguments. We then increment ‘all\_steps’ by the recursive steps, add the travel time between ‘start\_light’ and ‘second\_light’ to the time returned by the recursive call, and update ‘best\_tour’ and ‘best\_time’ if this time is faster than ‘best\_time’. After the for-loop we insert ‘start\_light’ at the front of ‘best\_tour’. Now that we have computed the fastest tour from the starting lighthouse through ‘L’, we can store that result in ‘memo’ using the ‘memo\_lookup\_key’. We than return ‘best\_tour’, ‘best\_time’ and ‘all\_steps’.

In all three cases the function returns the best tour, fastest tour time and number of steps taken by the program. We rely on the recursive and base case to solve the subproblems with varying starting lighthouses and ‘L’’s and use the memo hit case to avoid performing redundant work.

**Asymptotic bounds**

The runtime of the brute force approach without memoization is θ(n!) where n is the number of lighthouses in L. The runtime of the memoized version of \*fastest\_tour\_memo()\* can be investigating by tracking the number of subproblems encountered by the algorithm, as opposed to the brute force approach, and the complexity of each subproblem. The outer loop performed by the wrapper code resets memo and then kick-offs \*fastest\_tour\_memo()\* for each lighthouse in \*L\* as the starting lighthouse. This results in a factor of θ(n) for each function call. Within \*fastest\_tour\_memo()\* several constant time operations are performed, such as initialization of \*best\_tour\*, \*best\_time\* and \*all\_steps\* variables. Additionally, the constant time set operation \*list\_minus()\* is called to remove \*start\_light\* from \*L\*. We next construct the \*memo\_lookup\_key\* which involves creating a frozenset from \*L\*, which for the sake of analysis will assume is a constant time operation. A constant time dictionary lookup is performed to check if \*memo\_lookup\_key\* is in \*memo\*. If it is more constant time operations, such as dictionary insertion and addition performed before the algorithm returns. These operations can be compress into constant time factor of O(1) toward the runtime. Similarly, in the base case constant time operations, such as dictionary insertion and addition are performed before returning. In the recursive case several more constant time comparisons, additions and dictionary insertion are performed, as well as n-1 recursive calls. This contributes an $O(n-1)$ factor towards the runtime.

Each starting lighthouse has $(2^{n-1})$ unique subproblems to solve. This can be shown by viewing the problem in terms of how many unique combinations, of any size, of lighthouses can be made foreach set of lighthouses excluding the starting lighthouse. By equation C.4 in CLRS, the number of unique combinations of all sizes can be quantified as $(2^n)$. Since we have n starting lighthouses the total number of subproblems or sets of lighthouses searched with memoization is $n\*2^{n-1}$, we subtract one from \*n\* since we have locked the starting lighthouse into place. For each of these subproblems we perform $(n-1)$ recursive calls in the for-loop. Thus, in total the runtime can be quantified as:

$O((n-1)\*(n\*2^{n-1}+O(1)))→$

$O(n\*(n-1)\*2^{n-1})→$

$O(n^2\*2^n)$

**Retrospective**

Figure 1 nicely shows the increase in performance between the brute force implementation (red) and the memoized (blue) version of \*fastest\_tour\* algorithm design can have on a program’s execution – even if two functions are asymptoticly super polynomial, the difference in bounds can be dramatic. Figure 2 shows a zoomed in view of the asymptotic bounds of the brute force (red) and memoized (blue) algorithms. As noted above, the runtime of the brute force implementation is which can be nicely seen by the red dotted line. While the runtime analysis expected a tight bound of , the yellow dotted line, that represent this bound, does not present a tight bound on \*fastest\_tour\_memo\*, but an upper bound. Rather, we see the that the red dotted line representing better fits the number of steps taken by \*fastest\_tour\_memo()\*. This could point to a factor of *n* that was not taken into consideration during the runtime analysis or an implementation error. However, the function is bound from above by (black dotted line) as expected. One possible reason the memoized runtime deviates from the prediction is the use of starting light in the memo lookup key. Alternatively, the recursive function calls themselves should count as a step. Lastly, an implementation error or bug could be causing this error.

One thing that confused me was number of steps…

**Trying different base case….**

**Adding 1 to account for recursive work**

Figure 3 show the size of the memo dictionary, which is also the number of base cases and recursive cases taken by the program. As we see this almost perfectly aligns with the expected number of subproblems, Lastly, figure 4 shows the number of memo hits and misses that occur during the execution of \*fastest\_tour\_memo()\*.

One of the interesting parts of this assignment was managing the global memo dictionary. A utility function \*reset\_memo()\* was used to ensure that the memo dictionary was cleared before kicking off the code, and shared amongst single runs of the code, that is for all starting lighthouses. The choice of memo key structure was also important. An alternative to casting L to a tuple could be to construct a frozenset from L. A frozenset is an immutable data structure that tests for equality in an order independent manner. Surprisingly, the frozenset implementation did not have a difference in number of steps compared to the steps version.

Several code improvements were made, such as type hints, docstrings, error handling, wrapper functions and test cases. The test cases were especially useful in ensuring a proper implementation even when experimenting with memo lookup keys. Overall, this was a challenging lab but working through the algorithm on pen and paper, building helpful tables and charts and trying to see the problem from multiple perspectives, helped me get as close to a solution as possible.