Description of enhancement of 2-Opt Algorithm:

Initially, I decided to go over every line of code of my Basic algorithm to fine tune its efficiency. This resulted in a 68% increase in efficiency, reducing the average runtime of 5 runs of 200 trails of solving the same 100 city city set from 64s to 38s. This was achieved by making new variable containing the required submatrices of distance_matrix at the beginning of each of the respective for loops and calling this matrix instead of the entire matrix each time. This resulted in far less time spent retrieving and searching through large chunks of memory, as the matrix can be called millions of times, depending on the city set. Subsequently, as k-opt algorithms get stuck in local minima, in order to discover as many minima as possible, I decided to alter my algorithm to be able solve as many random un-optimised tours in a loop in a user-given time, returning the overall best tour found at the end. Furthermore, to the same end, I then decided to implement various different heuristics for deciding which edges to swap each visit to improving the given tour. After some experimentation with different heuristics, I observed that the best results, both tour wise and efficiency wise, came from a mixture of different heuristics, depending on the starting tour. I therefore decided to make my algorithm solve each given tour, using 8 different heuristics: Best-Change, First-Change, Worst-Change, Average-Change and 4 different editions of First-Change with a random chance each optimisation (2%, 5%, 10% and 20%) to select a random enhancement which has a negative effect. This random chance was added, in order to escape local minima, similar to how Simulated Annealing works. Unfortunately, even though all of these algorithms produced the best tours for certain tour configurations, due to time complexity and run-time, Worst-Change was dropped and the random First-Change editions were cut down to leave only the 5% variation. Finally, in order to further increase the number of tours checked in a given time, I decided to allow my algorithm to launch multiple python instances (set using the CPU_CORES variable) using a different core of the CPU each, each with their own loop, by implementing Multiprocessing. On a 4 core system, this would increase the efficiency of the algorithm by 300%.

Description of enhancement of Lin-Kernighan Algorithm:

Firstly, I decided to optimise the code of my original algorithm. I replaced the use of lists by sets where possible to reduce the complexity of the "in" operation from O(n) to O(1). I also reduced the number of validity checks throughout the algorithms various functions. After some experimentation and research, I decided that the original formulation of the algorithm from its paper was rather inefficient in terms of how it selected the edges to be swapped each iteration. I therefore decided to alter the algorithm to make the first valid k-opt move that it finds, rather than checking through all possible swaps and saving the best one. This resulted in an increase in efficiency of 91%, decreasing average runtime from 122s to 11s on trials on a 100 city city set. It is of note that this did not affect quality of results. Once optimised, I decided to experiment with some additional heuristics in order to increase the efficiency of my algorithm. After some further experimentation and research, I decided to limit my search of edges to add to just the nearest 5 neighbours of the latest node, vastly reducing the complexity of "deep-searches". To do this, the algorithm constructs a matrix containing this data before running, which is passed to each function, so that it is only computed once rather than once per call. I then decided implement two further heuristics, one for choosing yi and one for choosing xi (the edges to remove and add respectively). In choosing yi, I decided to give priority in order of increasing length. In the choosing of xi, I decided to check the longer edge before the shorter one. This is done under the philosophy that adding a smaller edge, while removing a larger one, is more likely to produce a better tour faster as it will optimise the amount by which the tour will be reduced, reaching its local minima first. These three heuristics combined resulted in an increase in performance of 60% on a 100 city city set. As LK is a variation on a k-opt algorithm, it suffers from the same issues of getting stuck in local minima, and therefore in order to discover as many local minima as possible, I decided to run my algorithm in a loop over as many randomised input as possible, returning the best tour found at the end. Finally, in order to further improve efficiency, I decided to implement Multiprocessing, setting up multiple loops, in different python instances, using their own CPU core each. I then decided to feed each of these loops a greedy tour starting at a different random city, as its initial starting tour as a final enhancement, as in testing this gave very promising results.