Three Different Implementations on Finding Neighbors in a Python List

Relevant Files

FindNeighbors.py – three different implementations of finding neighbors
PerformanceTesting.py – speed and scalability test of the three implementations
UnitTesting.py – stability test of three implementations

Introduction

This report compares three different implementations of generating neighbors in a Python list. The three different implementations optimize for different parameters: simplest to read and understand, fewest lines-of-code, and least peak memory. Performance test and unittest were run on each method to understand the speed, scalability, and stability of each implementation. Based on performance data and previous experience, it was found that the simplest to read and understand method is the preferred over all other implementation.

Input and Output

The input and output for all three methods are the same. The input is a Python list that may contain integers, strings, and objects and the output is a list of pairwise tuples, namely neighbors. The neighbors are defined as neighbor[i] = (input[i+1],input[i]). Duplicate tuples are allowed. An example is:
Input: [1, 3, 5, 7]
Output: [(3,1), (5,3), (7,5)]

Implementation

The simplest to read and understand (implementation A) method creates an empty output list and iterates through each input element sequentially. Neighbor pairs are created and appended onto the output list. The fewest lines-of-code (implementation B) shows the same implementation as A except the number of lines are reduced in order to retain space. The least peak memory (implementation C) creates pairs starting from the back of the input list. The pairs are inserted from the left of the output list in order to retain correct ordering of each tuple. Finally, the last element in the input list is popped off because it is no longer needed. Popping off elements helps reduce memory usage at the cost of speed due to O(n) insert operation.

Unittest and Performance Test

In order to validate the implementation multiple test cases such as empty list and integer list were tested. Lists containing integers, strings, and objects were also tested. The method assumes that a list will be inputted to the methods, otherwise a runtime error will occur.

In order to test speed and stability, multiple lists of random integers were generated and all the methods were timed. A scatter plot (Figure 1) shows that all implementations increase in time as input size increases. Implementation A and B exhibit linear behavior, while implementation C, fewest memory, exhibits exponential behavior. This is due to O(n) inserts in the while loop. Based on the data, it is advantageous to select the easily understood implementation or fewest lines-of-code because the linear speed is generally acceptable. However, the fewest lines-of-code is less readable. Consequently, a future developer may waste time on figuring out how the program works. It is easy to forget how you came up with solution further down the line when only optimizing for fewest lines of code. As result, easily understood implementation is the preferred method.

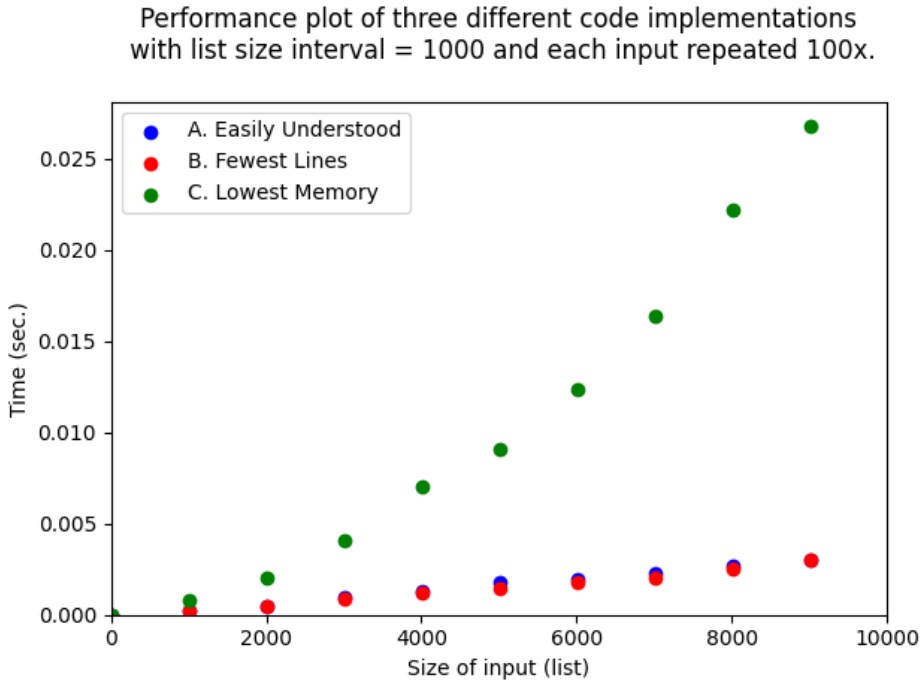

Figure 1: Scatter plot of speed of the three methods with respect to input size

Conclusion

This report compared three different methods of solving a problem, generating all neighbors given a Python list. The three methods, easily understood, fewest lines, and lowest memory, all have different tradeoffs. The lowest memory sacrifices exponential speed for memory while the other methods use more memory but have linear speed. Based on this report and prior software experience, I personally prefer the easily understood and readable code over the other implementations.