Alicia Sang
DCS229
Winter 2025

HW10 Reflection

a. This version uses a straightforward brute-force method. I loop through every possible pair of elements in the list and check whether their sum equals the target value. I do not use any additional data structures or create a new list. It is a simple approach that relies entirely on the original list, but it is not efficient for larger inputs. The algorithm is easy to implement, but not scalable.
b. Here, I use a dictionary to keep track of the values I have already seen. As I iterate through the list, I calculate what number would need to pair with the current one to reach the target. If that value is already in the dictionary, then a matching pair has been found. Otherwise, I add the current number to the dictionary and move on. This method takes advantage of constant-time lookups in dictionaries and significantly improves performance compared to the nested loop version.
c. This version uses a separate list as a presence map instead of a dictionary. It only works when the inputs are non-negative, since the indices of the list are used to represent values. I initialize a boolean list with length equal to $k + 1$, and as I go through the original list, I check whether the complement has already been seen. If so, I return true. If not, I mark the current value as seen. It mirrors the logic of the dictionary version, but trades generality for a slight gain in speed and simplicity in some cases.
d. The time complexity is $O(n^2)$ because the algorithm checks every pair of values in the list. There are no additional data structures, so the space complexity is $O(1)$. While this approach is easy to understand, it does not scale well and becomes slow as the input size increases.
e. The dictionary-based algorithm has a time complexity of $O(n)$ and a space complexity of $O(n)$. Each element is processed once, and each lookup or insertion into the dictionary takes constant time on average. This makes it much more efficient than the list-analysis version, especially for large datasets.
f. This version also runs in $O(n)$ time, since it only requires one pass through the list. However, the space complexity is $O(k)$, since it creates a new list of size $k + 1$. This is efficient when k is small, but can use a lot of memory if k is large. The method is fast and simple, but not as flexible as the dictionary approach.
g. When I tested the program with a list of 10,000 integers, the list-analysis version (List1) took noticeably longer than the other two. Both the dictionary-based and list-based methods completed very quickly, with runtimes measured in microseconds. If I increased the input size to 100,000, I would expect the list-analysis version to become much slower, while the other two would remain relatively efficient.

h. Even though the list-based method (List2) performed similarly in terms of timing, the dictionary approach is more flexible and scalable. The list-based method assumes all numbers are non-negative and within a known range, and it uses more space when k is large. The dictionary version handles negative numbers, avoids memory issues, and is more robust for general use cases.