Assignment 5 Analysis document:

- **Compare the running time of the push method. What is the growth rate of the method's running time for each stack class, and why?**

The push time for both stacks generally increases as N increases, which is expected due to the nature of the operations.
ArrayStack: The time grows relatively quickly for ArrayStack, especially for large values of NN. This is likely due to array resizing, which occurs when the array fills up. Resizing an array (especially doubling its size) is an O(N) operation, which can occasionally cause a spike in the time required to push elements. Since resizing happens less frequently, the time for ArrayStack is amortized O(1) in the average case, but large resizing operations can cause spikes in the timings.

The high value for ArrayStack Push Time for N = 200000 (4.73 million ns) is likely caused by such resizing events.

LinkedListStack: The LinkedListStack push times, while still increasing with NN, are noticeably lower than ArrayStack. This is expected because inserting an element into a linked list is always O(1), regardless of how many elements are in the list. The slight increase in time is likely due to the additional overhead of memory management and pointer manipulation for each new node.

- **Compare the running time of the pop method. What is the growth rate of the method's running time for each stack class, and why?**

ArrayStack: The pop operation for ArrayStack should be O(1), so its times should ideally remain constant as N increases. The initial low times are expected, and as the size increases, it should stay relatively constant.

The massive spike for N = 200000 (over 4 million ns) is highly unusual. Since popping from an array (just removing the top element) is an O(1) operation, this spike suggests that something outside the actual pop operation (e.g., memory fragmentation or a measurement error) might be affecting the results.

LinkedListStack: The pop operation for the linked list is also O(1). The observed times seem increasingly erratic. While the times do increase somewhat with N, the performance drop at N = 100000 (1.34 million ns) is unexpected and could indicate inefficiencies with the linked list (e.g., memory allocation overhead, pointer chasing). Linked lists are generally slower than arrays due to these overheads, but the times

shouldn't have such large spikes unless something is affecting the memory access pattern or the implementation.

- **Compare the running time of the peek method. What is the growth rate of the method's running time for each stack class, and why?**

ArrayStack: The peek operation for the array stack should also be O(1), as it just accesses the top element of the array. The observed times seem to be increasing with N, which is unexpected. The peek operation should not show much variation with increasing NN. The large increase for N = 100000 (1.2 million ns) and N = 200000 (2.5 million ns) suggests that there may be other factors at play (e.g., memory fragmentation, additional overhead in the peek method, or measurement error).

LinkedListStack: The peek time for the linked list is similarly increasing with NN, but the increase is less pronounced than in ArrayStack. The increase is expected, but the time for N = 200000 (2.7 million ns) is still somewhat high. Again, this may be caused by memory overhead associated with managing a linked list structure.

- **Based on your timing experiments, which stack class do you think is more efficient for using in your WebBrowser application? Why?**

LinkedListStack can be more efficient if you're frequently pushing and popping elements and want to avoid occasional resizing costs. Since LinkedListStack maintains a constant time O(1) for all operations, it would be a safer choice if you anticipate frequent stack operations with a variable number of items.