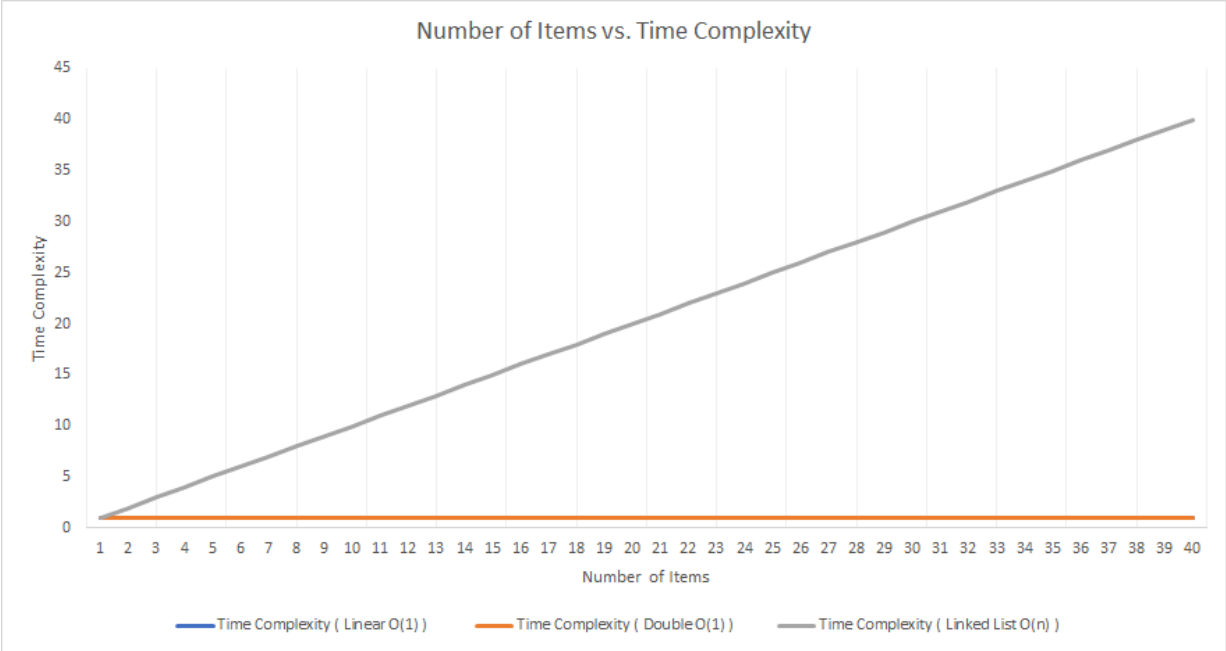
PA1 Report
Otokini Cotterell
UIN 233001684

Introduction. The objective of this assignment was to understand how stacks are implemented, how the data structure holds data, and the algorithmic complexity involved in storage, retrieval, and search capabilities—the assignment involved using stacks with an array and a linked list.

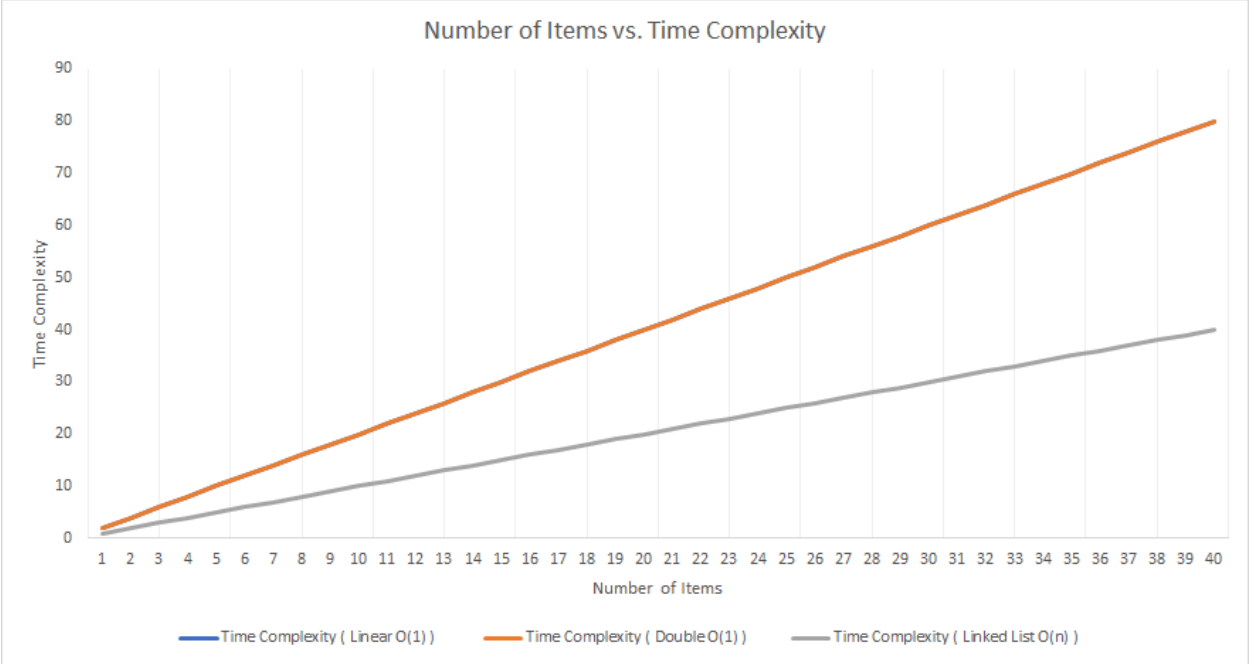
Theoretical Analysis. The algorithmic complexity of pushing an item onto a stack is O(1) in terms of the isolated push operation. However, depending on the additional data held in the stack, it might take O(n) operations to obtain the proper location where the item should be pushed. In the linked list example, it was required to iterate through the entire linked list before getting to the required tail node. Such could be avoided completely by holding an extra pointer to the tail node. In the array, because the top_index was always held, it took O(1) operations to search for the right location. However, if the array had to be resized, the time to do the proper resizing had an algorithmic complexity of O (2n) (n operations for copying, n operations for deleting old array).

With the linear stack, resizing operations were done more often but held less memory and were more true to the size required at the current moment. It could be a serious downside if the stack has a fast growth rate. The double stack did much less resizing operations, but often required much more memory than what was needed by the stack. This method could be a serious downside if the stack is very slow to grow and doesn't need as much memory as the doubled stack is allocating. The linked list stack was always true to size, but to push an item onto the stack O(n) operations always had to be done to push. Though the entire stack had to be iterated to push an item, resizing algorithms were not needed.

Experimental Results.



The time complexity of the linear stack is directly under the double stack. It is important to note that this is solely the performance of the push function. Wen considering the resizing functions for each stack, the time complexity becomes a function that more follows this:



- When making bigger sized stacks, linked lists have a more consistent big O time complexity than arrays that need to resize.
- If the stacks need to resize often, and grows quite often, a stack that doubles its memory is more efficient than a stack that only increases its memory by a constant.

- If the stack doesn't need to resize often, and doesn't use too much memory and/or memory is critical to the program's functionality, a stack that grows by a constant is typically better.

Overall, if memory is not an issue, a double resizing stack is likely faster than the other two options. For discrepancies between the theoretical and experimental analysis, one important thing to note is that the double and linear stacks do not resize every time an element is added unto the stack. It only resizes when the stack is already at maximum capacity. Consequently, the time complexity of these stacks (except for the linked list function) is much more complex than theorized, and depends on the starting max capacity of the stack. In essence, pushing an element unto the stack both StackLinear and StackDouble could potentially have a big O notation of O(1) for the first 5 pushes, then O(2n) for the 6th push. The linked list stack always has a push time complexity of O(n), so the theoretical analysis is much like experimental analysis.