1. **In Array_Deque:** __init__(self), __len__(self), pop_front(self), peek_front(self), pop_back(self), and peek_back(self) has O(1) as their worst-case performance. There is not any loop implemented inside them. Each of them has unchanged steps of implementation. __grow(self) has O(n) as its worst-case performance. __grow(self) basically initiates a new array with doubled size and copies all the eligible elements from the old array by iterating from front to back, so its performance also depends on the scale n and is O(n). push_front(self) and push_back(self)'s worst-case performances are also O(n), because they include __grow(self). When the current self.size reaches the self.capacity, they need to first execute __grow(self) to increase the size, which is the worst case and is O(n). __str__(self) goes through every element inside the array between front and back and concatenate the element to a string, which is linear time, so it has quadratic performance.
   In Linked_List_Deque: O(1) is the worst-case performance for every method except __str__(self). __len__(self) simply return the value of size. All the push, pop, and peek methods implement append, remove, or insert methods from the Linked_List class. append is obviously O(1). Remove and insert are also O(1) in push, pop and peek because they only remove or insert at the first index or the last index of the linked list. In the Linked_List class, removing or inserting at front or back is of constant performance because they don't need to iterate through the list, but just directly go to the front or back. Therefore, all the push, pop, and peek methods in Linked_List_Deque has O(1) as its worst-case performance. __str__(self), the same as that in Array_Deque, has O(n^2) performance.
2. In Array_Deque, I distinguish between an empty deque and a deque with one entry simply by checking the size of the self.size and the index of front and back. If self.size is 0 and self.front=self.back, the deque is empty. Otherwise if self.front is not equal to self.back, the deque has at least one entry.
3. Because if the grow method only increases the size by one, it will be more frequently called because the capacity will be more easily reached. However, every time __grow(self) is invoked, it needs to initiates a new array and copy the original values into it, which is of O(n) performance. So it will be low-efficient for __grow() to only increase the size by one. If __grow() doubles the size, as the capacity n increases, it will be less easily for the deque to reach the capacity and allows for more space for new entries. __grow() will be less frequently to be called.
4. In my DSQ_Test.py, almost all the functions test all of deque, stack and queue (for brevity) together because they share some similar attributes. I test the situations that: printing deque while empty, length while empty, printing deque after pushing_front one entry, printing deque after pushing_back one entry, length while there is one entry, popping while empty, popping front and back while there is one entry, peeking front and back while there is one entry, length while there is 3 entries, printing deque after pushing 3 different elements including a None, printing deque after pushing 3 elements and popping one from front or back, peeking from back and front while there is 3 entries, and popping front and back when empty. They cover all expected possibilities and tests all the methods implemented.