bag_da.py (https://canvas.oregonstate.edu/courses/1826035/files/83413640?wrap=1) (https://canvas.oregonstate.edu/courses/1826035/files/83413640?wrap=1)

stack_da.py (https://canvas.oregonstate.edu/courses/1826035/files/83413641?wrap=1) (https://canvas.oregonstate.edu/courses/1826035/files/83413641?wrap=1)

queue_da.py (https://canvas.oregonstate.edu/courses/1826035/files/83413642?wrap=1) (https://canvas.oregonstate.edu/courses/1826035/files/83413642?wrap=1)

static_array.py (https://canvas.oregonstate.edu/courses/1826035/files/83451506?wrap=1) (https://canvas.oregonstate.edu/courses/1826035/files/83451506?wrap=1)

(https://canvas.oregonstate.edu/courses/1826035/files/83132499/download?wrap=1)

Part 5: Amortized Analysis

Consider the append**()** operation for a Dynamic Array. In the best case, the operation is *O(1)*. This corresponds to the case where there was room in the space we have already allocated for the array. However, in the worst case, this operation slows down to *O(n)*. This corresponds to the case where the allocated space was full and we must copy each element of the array into a new (larger) array. This problem is designed to discover runtime bounds on the average case when various array expansion strategies are used, but first, some information on how to perform an amortized analysis is necessary.

1. Each time an item is added to the array without requiring reallocation, count 1 unit of cost. This cost will cover the assignment which actually puts the item in the array.
2. Each time an item is added and requires reallocation, count X + 1 units of cost, where X is the number of items currently in the array. This cost will cover the X assignments which are necessary to copy the contents of the full array into a new (larger) array, and the additional assignment to put the item which did not fit originally.

To make this more concrete, if the array has 3 spaces and is holding 2 items, adding the third will cost 1. However, if the array has 3 spaces and is holding 3 items currently, adding the 4th item will actually cost 4 (3 to move the existing items + 1 to assign the 4th item once space is available).

Now our goal is to calculate how many "units" are spent in the entire sequence of N append operations then use it to find the average "unit" cost for an append. When we can bound an average cost of an operation in this fashion, we call it amortized execution time. Note that this method charges the same amortized execution time to each operation in the sequence of N append operations even though the same operation will experience very different costs across a sequence. It relies on the fact that many of the operations in the sequence contribute little cost and only a few operations contribute a high cost to the overall time.

In a file called **amortizedAnalysis.pdf,** please provide answers to the following questions:

1. How many cost units are spent in the entire process of performing 40 consecutive append operations on an empty array which starts out at capacity 3, assuming that the array will *double in capacity each time* a new item is added to an already full dynamic array? As N (i.e., the number of appends) grows large, under this strategy for resizing, what is the amortized big-O complexity for an append?
2. How many cost units are spent in the entire process of performing 40 consecutive append operations on an empty array which starts out at capacity 3, assuming that the array will *grow by a constant 2 spaces each time* a new item is added to an already full dynamic array? As N (i.e., the number of appends) grows large, under this strategy for resizing, what is the amortized big-O complexity for an append?

You must show detailed work to calculate the total cost when the number of append operations is 'N' (for both cases). You are highly suggested to use the format provided here: amortizedAnalysis_format.pdf (https://canvas.oregonstate.edu/courses/1826035/files/83471991?wrap=1) (https://canvas.oregonstate.edu/courses/1826035/files/83471991?wrap=1)

The above file in .docx format: amortizedAnalysis_format.docx (https://canvas.oregonstate.edu/courses/1826035/files/83649109?wrap=1) (https://canvas.oregonstate.edu/courses/1826035/files/83649109?wrap=1)

Scoring

- Implementation of Dynamic Array and ADTs using Dynamic Array[100 pts]

Dynamic Array:

append() (6 points)
insert_at_index() (6 points)
remove_at_index() (6 points)
resize() (8 points)
slice() (8 points)
merge() (8 points)
map() (6points)
filter() (6 points)
reduce() (6 points)----(60 points)

Bag:

add() (5 points)
remove() (5 points)
count() (3 points)

clear() (2 points)
equal() (10 points) -----(25 points)

Stack:

push() (3 points)
pop() (3 points)
top() (3 points) -----(9 points)

Queue:

enqueue() (3 points)
dequeue() (3 points) -----(6 points)

- Amortized Analysis [20 pts]

What to Turn In (4 files)

You will turn in the above four files via both Canvas and Gradescope. The written analysis will go to Canvas, the other files will go to Gradescope.

The analysis file, amortizedAnalysis.pdf, will be turned in through this Canvas assignment: **Assignment 2: Written Analysis** **(https://canvas.oregonstate.edu/courses/1826035/assignments/8193339)**

The python files will be submitted through Gradescope from this assignment: **Assignment 2: Gradescope Submission** **(https://canvas.oregonstate.edu/courses/1826035/assignments/8193340)**

1. dynamic_array.py
2. bag_da.py
3. stack_da.py
4. queue_da.py