DSC 190 - Homework 01

Due: Wednesday, January 20

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope on Wednesday at 11:59 p.m.

Problem 1.

We saw in lecture that Python's `list` is a dynamic array. In this problem, we will empirically time operations on `list`s and reconcile the results with the theoretical time complexities we have derived.

a) We will first study `.append`. Create a plot in which the independent variable is the array size, n, and the dependent variable is the time taken by `.append` on an array of size n. To reduce the effect of noise, compute the time taken by `.append` on an array of size n by performing 100 trials and averaging the timings. The independent variable, n, should range from 1 to 1000. Comment on whether your plot agrees with theory, which predicts that the worst case complexity is $\Theta(n)$, but the amortized complexity is $\Theta(1)$. Include your code and your plot.

Hint: we must be very careful when performing these timings to avoid optimizations by the memory allocator. The recommended way to perform this timing is to create a function `time_append(n)` which: 1) creates 100 lists of size n (their exact contents does not matter), 2) appends an arbitrary element to each list, timing each append, and 3) returns the average time taken. By creating 100 distinct lists, we make it difficult for the memory allocator to optimize by reusing a list that was recently deallocated.

b) Python `list`s use a somewhat strange growth strategy. While the growth is geometric, the growth parameter is quite small, and there is an extra constant term. To be precise, for all but the smallest arrays, Python uses the following rule:

$$(\text{new size}) = \gamma \times (\text{old size}) + 6$$

Using the result of the previous part, what is the growth factor γ? There is a value of γ that predicts the next size in the sequence *exactly* (after perhaps taking the floor or ceiling of the new size).

c) Let's analyze popping from the *end* of the array. Repeat the analysis of part (a), timing `.pop()` instead of `.append`. Comment on whether your plot agrees with theory. You do not need to include your code, but include your plot.

d) Next let's analyze popping from the *start* of the array. Repeat the analysis of part (a), timing `.pop(0)` instead of `.append`. You do not need to include your code, but include your plot. Comment on whether your plot agrees with theory.

e) What operating system and version of Python are you using to calculate your timings? It is possible that the behavior of `list` is dependent on these.

Programming Problem 1.

In a file named `min_heap.py`, implement a `MinHeap` class. Your class should have the following methods:

- `.min()`: return (but do not remove) minimum key
- `.decrease_key(i, key)`: reduce the value of node i's key to `key`

- `.insert(key)`: insert a new key, maintaining the heap invariant
- `.pop_min_key()`: remove and return the minimum key

Programming Problem 2.

In a file named `online_median.py`, create a class named `OnlineMedian` which has two methods: `.insert(x)`, which inserts a number into the data structure in $O(\log n)$ time, and `.median()` which computes the median of all numbers inserted so far in $\Theta(1)$ time.

Programming Problem 3.

In a file named `is_heap.py`, create a function named `is_heap(arr)` which accepts a non-empty numpy array `arr` and checks to see if it is a binary max heap, returning `True` if it is and `False` if it isn't.