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.

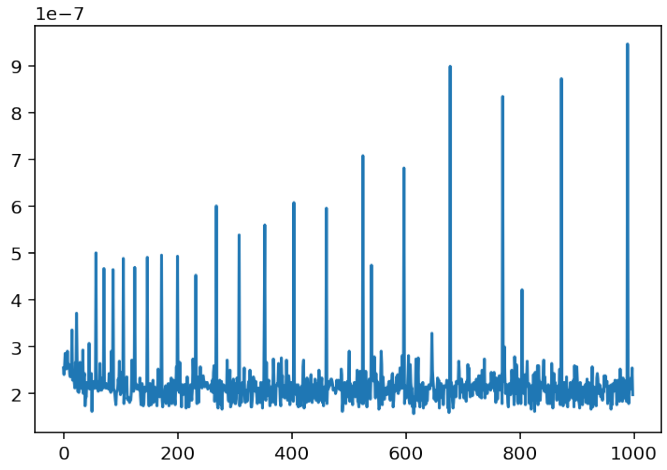
Solution: Here is the code I used to time appends:

```python
import time
import numpy as np
import matplotlib.pyplot as plt

def list_of_size(n):
    """Create a dummy list of size n"""
    lst = []
    for i in range(n):
        lst.append(i)
    return lst

def timer(f, *args):
    start = time.time()
    f(*args)
    stop = time.time()
    return stop - start

def time_list_operation(operation, n, k=100):
    lists = [list_of_size(n) for _ in range(0, k)]
    times = [timer(operation, lst) for lst in lists]
    return sum(times) / k

def append(lst):
    lst.append(0)
```

```
K = 1000
append_times = np.array([time_list_operation(append, i) for i in range(1, K)])
```

The code is written so as to make it easy to time other list operations, such as pops, later on.

Notice how we're creating a list of size n. The first way you might try to do this is by writing something like `[0] * n`. If we do this, however, we're unlikely to see the expected result when we make our plot. I'll describe why in a moment. But first, the plot...



We see a bunch of regularly-spaced "spikes" – these correspond to the physical size of the dynamic array being increased.

If we had used `[0] * n` or similar to create our dynamic array of size n, we might not see these regularly-spaced spikes. Why? When we write `[0] * n`, Python creates a new dynamic array, allocating at least enough space to hold n numbers. In fact, it (probably) *over-allocates*, since there's a good chance that you'll be appending elements soon. Therefore, we can guess that the physical size of the dynamic array will be larger than n, and so appending an element is cheap (there's always an empty space).

b) Python `lists` 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).

Solution: If we take the array of times from the previous part, we can infer the growth rate. Because there is some noise, we'll only consider the spikes that are above 6×10^{-7} in height. This code does the trick: `np.where(append_times > 6e-7)` The result is (for me): 678, 770, 873, 989. You probably got similar.

If we take the ratio of consecutive pairs (subtracting 6 from the larger size as the problem statement suggests), we get something close to 1.125. For instance, $(770 - 6)/678 \approx 1.1268$.

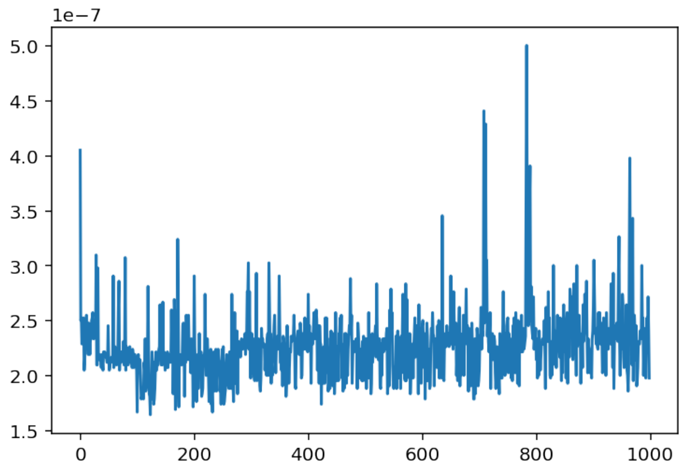
We might guess that the real growth factor is 1.125 and that some rounding is causing the difference we see. In fact, you can check that it appears that Python is rounding (or perhaps taking the ceiling), so that the new size is given by:

$$\lceil 1.125 \times (\text{old size}) + 6 \rceil$$

For example, $1.125 \times 770 = 872.25$, taking the ceiling gives 873, as expected.

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.

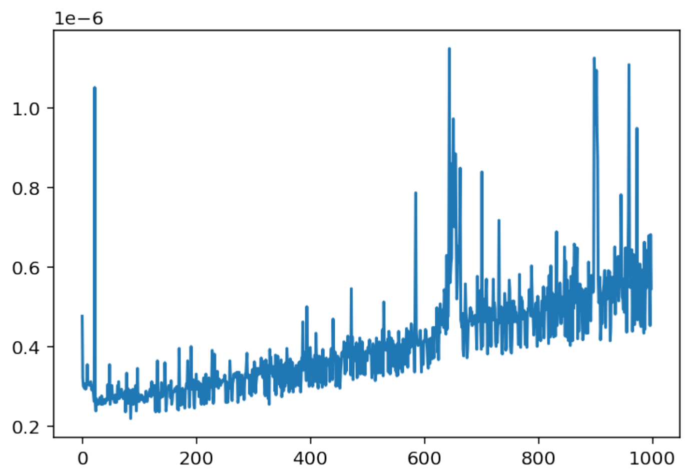
Solution: Here is the plot I see.



Theory predicts that `.pop` takes constant time, although this is assuming that the implementation is doing something reasonable. This plot looks fairly flat, though there are some interesting spikes. These spikes could be due to the memory manager moving the block in order to defragment the memory.

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.

Solution: Here is the plot I see.



Theory predicts that `.pop` takes linear time in the size of the list, and the plot seems to support this.

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.

Solution: I ran these experiments on NixOS Linux kernel 5.4.72 with Python 3.8.5 compiled with GCC 9.3.0.

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

Solution: Here's the straightforward implementation of a min heap. It is a modification of the code for a max heap that was given in lecture.

```python
def parent(ix):
    return (ix - 1) // 2

def left_child(ix):
    return 2*ix + 1

def right_child(ix):
    return 2*ix + 2


class MinHeap:

    def __init__(self, keys=None):
        if keys is None:
            keys = []
        self.keys = keys

    def min(self):
        return self.keys[0]

    def _swap(self, i, j):
        self.keys[i], self.keys[j] = self.keys[j], self.keys[i]

    def decrease_key(self, ix, key):
        if key > self.keys[ix]:
            raise ValueError('New key is larger.')

        self.keys[ix] = key
        while parent(ix) >= 0 and self.keys[parent(ix)] > key:
            self._swap(ix, parent(ix))
            ix = parent(ix)

    def insert(self, key):
        self.keys.append(key)
        self.decrease_key(len(self.keys)-1, key)

    def pop_min_key(self):
        if len(self.keys) == 0:
            raise IndexError('Heap is empty.')
        lowest = self.min()
        self.keys[0] = self.keys[-1]
        self.keys.pop()
        self._push_down(0)
        return lowest

    def _push_down(self, i):
        left = left_child(i)
        right = right_child(i)
        if left < len(self.keys) and self.keys[left] < self.keys[i]:
            smallest = left
        else:
            smallest = i

        if right < len(self.keys) and self.keys[right] < self.keys[smallest]:
            smallest = right

        if smallest != i:
            self._swap(i, smallest)
            self._push_down(smallest)
```

A clever alternative is to implement a min heap by wrapping a max heap and negating keys as you insert them.

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.

Solution: As discussed in lecture, we can implement this with a min heap and a max heap. The min heap stores the upper half of the data, while the max heap stores the lower half.

As we insert a key, we need to compare it to the tops of the heaps. If it is smaller than the max of the lower heap, we insert into it. Otherwise, we insert into the upper heap.

If the heaps become unbalanced we should rebalance them. This is as simple as popping an element from the larger heap and inserting it into the smaller heap.

The following code assumes that you've also copied the implementations of `MaxHeap` and `MinHeap` into the same file.

```python
class OnlineMedian:
    def __init__(self):
        self._upper = MinHeap()
        self._lower = MaxHeap()

    def insert(self, key):
        if self._upper:
            if key >= self._upper.min():
                self._upper.insert(key)
            else:
                self._lower.insert(key)
        else:
            self._lower.insert(key)

        if len(self._lower) - len(self._upper) >= 2:
            self._rebalance(self._upper, self._lower)
        elif len(self._upper) - len(self._lower) >= 2:
            self._rebalance(self._lower, self._upper)

    def median(self):
        if len(self._lower) == len(self._upper):
            return self._lower.max()
        elif len(self._upper) > len(self._lower):
            return self._upper.min()
        else:
            return self._lower.max()

    def _rebalance(self, smaller, larger):

        def pop(container):
            if isinstance(container, MinHeap):
                return container.pop_min()
            else:
                return container.pop_max()

        while len(larger) - len(smaller) >= 2:
            x = pop(larger)
            smaller.insert(x)
```

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.

Solution: To check if an array represents a binary max heap, we only need to check the heap property (the completeness property is assumed). To do this, we recursively check that each node's key is $\geq$ than the keys of its children:

```python
def parent(x):
    return (x - 1) // 2

def left(x):
    return 2 * x + 1


def right(x):
    return 2 * x + 2


def is_heap(arr, node_ix=0, parent_key=float('inf')):
    if node_ix >= len(arr):
        return True

    node_key = arr[node_ix]

    return (
        (node_key <= parent_key)
        and
        is_heap(arr, left(node_ix), node_key)
        and
        is_heap(arr, right(node_ix), node_key)
    )
```