Class Activity 1: Comparing the Run Times of Common Python Data Structures

Learning Outcomes

- Review common data structures and the run times of their operations
- Become familiar with built-in Python data structures
- Learn how to write benchmarks in Python

Introduction

In the Data Structures, you learned about a number of data structures such as ArrayLists, LinkedLists, Stacks implemented with ArrayLists, Queues implemented with LinkedLists, Hash tables, and Binary Search Trees. You also learned how to model the asymptotic run time of the operations (e.g., contains, add, and remove) associated with these data structures using big-O notation.

In this class, we are going to use Python with Jupyter Notebooks. These notebooks will allow you to include plots alongside your implementations of algorithms and data structures.

The data structures and algorithms you learned are not specific to Java, however, and are also applicable to Python. Most programming languages include similar data structures and the caveats about the run times of their operations apply. Python contains several built-in data structures:

- list: a growable list backed by an array. Similar to Java's ArrayList.
- dictionary: a map implemented with a hash table. Similar to Java's HashMap.
- set: a set implemented with a hash table. Similar to Java's HashSet.

The Python standard library contains additional data structure implementations such as:

- Collections.deque: a double-ended queue implemented as a doubly-linked list which maintains references to the head and tail nodes. Similar to Java's LinkedList.
- heapq: a priority queue implemented using a heap data structure. (We'll cover heaps later in this class).

The Python and its standard library also implement algorithms for common operations:

- sorted(iterable): Returns a sorted list containing the elements in iterable.
- list.sort(): Sorts the items in the list object.
- bisect: The bisect module provides implementations of binary search operations.

In this exercise, you are going to practice benchmarking in Python by comparing operations on lists and deques.

Instructions

You may use either a Jupyter Notebook running on your laptop or ROSIE this quarter. Nothing we will do will require GPUs or more compute resources than your laptop has. Should you choose to work on your laptop, the easiest way to setup a Python Notebook environment is with the 64-bit Anaconda Distribution (https://www.anaconda.com/products/individual). Should you choose to use ROSIE, you can access the ROSIE documentation here: https://msoe.dev/.

Problem 1: Benchmark the append() operation on Lists and Deques

The Python method for adding an element to the end of a data structure is append() and is equivalent to Java's add(). Read the documentation for the Python deque class.

You can access the deque class by importing it:

```
from collections import deque
```

You can benchmark a block of code with the following template:

```
import time

# DO ANY SETUP

start_time = time.perf_counter()
for i in range(REPETITIONS):
   # PUT CODE YOU WANT TO BENCHMARK HERE
end_time = time.perf_counter()
elapsed = end_time - start_time
```

In the end, elapsed will contain the elapsed time in units of seconds.

Benchmark the time required to add 1,000,000 items to a list and a deque and compare the run time. In both cases, you should start with empty data structures. At the end of the benchmark, each data structure should have 1 million items. You can simply use a string or integer constant as the item.

Problem 2: Benchmark the insert(0, ITEM) operation on Lists and Deques

The Python method for inserting an element at an arbitrary positions is insert(idx, item). If there is an item already at that index, that item and everything following it will be shifted to the right to make room for the new item.

Benchmark the time required to insert 1,000,000 items at the front of lists and deques and compare the run time. In both cases, you should start with empty data structures. At the end of the benchmark, each data structure should have 1 million items. You can simply use a string or integer constant as the item. (You can expect this to take a few minutes to run.)

Problem 3: Compare the "in" (contains) operation on Lists and Sets

The Python syntax for determining if a data structure contains an item is as follows:

```
item in data_structure
```

The expression will return a Boolean value.

Benchmark the time required to call "in" 100,000 times on lists and sets with 100,000 items. In both cases, you add 100,000 unique items to an empty set and an empty list before the benchmark. It is simplest to add the integers from 1 to 100,000. To capture the worst case run time, you should check for an item (e.g., -5) that is NOT in the data structures. (Expect this to take a few minutes.)

Reflection Questions

a. Create a table using Markdown syntax of the run times from the benchmarks. The table should have 3 columns: the operation, the first data structure, and the second data structure. Each cell should contain the run-time in seconds. Make sure the table includes a header.

b. In which cases were the run times approximately similar versus different?

c. Add two more columns to your table. In these columns, put the big-o notation for the operations for those data structures based on what you remember from CS 2852.

d. Are the ordering of the run times consistent with the ordering based on big-o notation?