CS 3851 Algorithms - Lab 1: Comparing the Run Times of Common Python Data Structures

Stuart Harley

Introduction

In this lab we review the common data structures list, deque, and set, and the run times of some of their operations. Specifically the append, insert (at index 0), and "in" (contains) operations. In order to better visualize the time complexities of these operations, we are going to benchmark these operations.

Importing Libraries

In [1]:
```
from collections import deque
import time
```

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

Starting with an empty list, benchmarking the time required to add 1,000,000 items to the list. Using the number 5 as an integer constant to be added.

In [2]:
```
l = []
start_time = time.perf_counter()
for i in range(1000000):
    l.append(5)
end_time = time.perf_counter()
print(end_time - start_time, 'sec')
```

```
0.08102197502739727 sec
```

Starting with an empty deque, benchmarking the time required to add 1,000,000 items to the deque. Using the number 5 as an integer constant to be added.

```
In [3]: d = deque()
        start_time = time.perf_counter()
        for i in range(1000000):
            d.append(5)
        end_time = time.perf_counter()
        print(end_time - start_time, 'sec')
```

```
0.08276941103395075 sec
```

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

Starting with an empty list, benchmarking the time required to add 1,000,000 items at the front of the list. Using the number 5 as an integer constant to be added.

```
In [4]: l = []
        start_time = time.perf_counter()
        for i in range(1000000):
            l.insert(0, 5)
        end_time = time.perf_counter()
        print(end_time - start_time, 'sec')
```

```
268.5798508269945 sec
```

Starting with an empty deque, benchmarking the time required to add 1,000,000 items at the front of the deque. Using the number 5 as an integer constant to be added.

```
In [5]: d = deque()
        start_time = time.perf_counter()
        for i in range(1000000):
            d.insert(0, 5)
        end_time = time.perf_counter()
        print(end_time - start_time, 'sec')
```

```
0.11011907400097698 sec
```

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

Adding 100,000 unique items to an empty list. Using the numbers 1 through 100,000 for this. Then benchmarking the time required to call "in" 100,000 times on a list. In order to capture the worst case run time, checking for a item not in the list, -5.

```
In [6]: l = []
        for i in range(100000):
            l.append(i+1)
        start_time = time.perf_counter()
        for i in range(100000):
            -5 in l
        end_time = time.perf_counter()
        print(end_time - start_time, 'sec')
```

```
83.21763147902675 sec
```

Adding 100,000 unique items to an empty set. Using the numbers 1 through 100,000 for this. Then benchmarking the time required to call "in" 100,000 times on a list. In order to capture the worst case run time, checking for a item not in the list, -5.

```
In [7]: s = set()
        for i in range(100000):
            s.add(i+1)
        start_time = time.perf_counter()
        for i in range(100000):
            -5 in s
        end_time = time.perf_counter()
        print(end_time - start_time, 'sec')
```

```
0.004833626910112798 sec
```

Reflection Questions

a. Creating a table of the run times from the benchmarking. All times are in seconds.

Operation	List Time	Deque Time	Set Time
append()	0.081	0.083	
insert(0, x)	268.560	0.110	
in	83.218		0.005

b. The append cases run times were approximately similar.

The insert(0,x) and "in" cases run times were different.

c. Adding the big-o notation for these operations to the table.

Operation	List Time	Deque Time	Set Time	List big-o	Deque big-o	Set big-o
append()	0.081	0.083		O(1)	O(1)	
insert(0, x)	268.560	0.110		O(n)	O(1)	
in	83.218		0.005	O(n)		O(1)

d. The ordering of the run times are consistent with the ordering based on big-o notation.

Conclusion

When choosing a data structure when writing a program, it is important to pick a structure that is efficient at the most common operations that the program will need. Different data structures has different time complexities for common operations, so picking the wrong structure can greatly slow down the exectution of your program. We visualized this above with the append, insert, and in operations. The differences between a O(1) operation from one data structure and the same operation that is O(n) on a different data structure can be massive depending on the amount of data.

For example, inserting into the front of a deque is constant time and took .11 seconds since the deque is built using a double-linked list. While inserting into the front is linear time and took 268.56 seconds since every value in the list had to be shifted to be able to insert at the front. This operation took 2441 times longer running on the list compared to the deque.