COMP 1405Z
Fall 2023 – Tutorial #5

Objectives

- Implement a hybrid data structure to improve runtime complexity
- Use binary search to improve counting speed in a sorted list

Problem 1 – Improving Runtime Complexity

Dictionaries are very nice because they let us insert, remove and determine if a key is present in O(1) time (i.e., the time does not depend on the number of keys in the dictionary). They do not, however, allow us to maintain the order that the keys are added, which may be important in certain applications, like when time/age of data is involved such as the 'storing sensor data' problem from a previous set of practice problems. Lists allow us to insert/remove in O(1) time (it actually depends on how they are implemented, but we will assume this is true here) and maintain the order of inserted elements, but determining whether an element is in an unsorted list takes O(n) time.

For this question, you will implement an improved queue-like data structure that uses more memory space (i.e., stores more data), but allows us to insert, remove, *and* determine if an item is present in constant time, while also maintaining the order of item insertion. To start, open the improved-queue.py file from the tutorial page on Brightspace. This file has add and remove functions that modify a list variable by adding an element to the end or removing an element from the start (like a queue). The provided solution maintains the order of items as they are inserted, but the containslinear search function will run in O(n) time. You must modify this file by adding an additional function, called containshash(dict, value), that will determine if an item exists in the list in O(1) time. The original list-based functionality should be maintained.

This will require you to store additional information in a dictionary. Additionally, you will have to add additional arguments to your function to pass the dictionary variable as input. Your final functions should look like: addend(list, dict, value), removestart(list, dict), containslinear(list, value), and containshash(dict, value).

If you want to test your implementation, you can copy/paste the code from the listandhash-test1.txt and listandhash-test2.txt files into your program. The first piece of testing code will randomly add/remove some values before printing out the list and hash. This will allow you to verify that things seem to be working. The second piece of testing code will perform significantly more operations and verify that your containshash function always returns the same result as the containslinear function. It will also compare the search time using the linear function to the search time using your hash-based function.

Problem 2 - Binary Search Counting

Assume you are given a sorted list and need to create a function that takes a value as input and returns how many times that value occurs in the list. You could accomplish this using a linear search approach that would take O(n) time. Instead, you can use a binary search approach to find the first index that the value occurs at and a second binary search to find the last index the value occurs at. This will allow you to determine how many times the item occurs in the list in O(log n) time.

Write a program in a file called **binarycount.py** that has 3 functions:

1. **count(list, value)** – returns the number of times **value** occurs in the sorted **list** by using the following two functions to determine the start and end index of the specified value.
2. **findstart(list,value)** – returns the index representing the first occurrence of **value** in the sorted **list**. This should use a modified binary search process – instead of comparing the value at the index to the value you are looking for, you must determine if the index is the first occurrence of the value (i.e., item at index == value and item at index-1 != value). You still should be able to decrease the search space by ½ on each iteration.
3. **findend(list,value)** - returns the index representing the last occurrence of **value** in the sorted **list**. This should use a binary search process similar to the one described for the previous function, but you will again have to change what you are searching for (the last index, in this case).

Using the listandhash-testX.txt files as a guide, use the time module to measure the difference in execution time between your binary counting solution and the built-in linear list.count() function that Python provides. Your code should generate a large list (~2500 items) of random numbers to search in (**X**), as well as a large list of random numbers (~50000) to search for (**Y**). Note, you can start with smaller lists initially if you want to verify your solution is working first. Ensure that **Y** has a mix of numbers that are present in **X**, as well as numbers that are not present in **X**. You must then:

1. Perform a binary count in **X** and a linear list.count() in **X** for each number in **Y**. Ensure that the count values for each value are equal for both functions to verify that your binary count function is working correctly.
2. Record the time it takes to perform a linear list.count() in **X** for each number in **Y**.
3. Record the time it takes to perform a binary count in **X** for each number in **Y**.
4. Print out a comparison between the two counting functions.

In comments within your code file, write a short analysis. Some questions to consider?

1. Is your binary search count faster than the built-in list count?
2. What happens if you change the size of **X** (i.e., how does the time difference change for smaller/longer lists)?
3. What if you change the range of the numbers added to **X** so there are more/less duplicates?
4. What if a larger proportion of numbers in **Y** are not present in **X**?
5. Do these results make sense given what you know about the runtime complexity of both functions?

Submission

Add each of your code files and the testing resources to a single .zip file named "**tutorial5.zip**" and submit it to Brightspace. Ensure that your file is a .zip and has the proper name. Download your submission afterward and check the contents to ensure everything works as expected.