COMP 1005/1405
Summer 2017 – Tutorial #5

Tutorial Objectives

- Practice writing code to **search and minimize run-time complexity**, **writing recursive code**, as well as the use of other control structures and data types we have used previously.
- If you have not compelted the questions from last week's tutorial because we had not covered all of the material in time, you should work on some of those questions first.

Problem 1

'Galloping search' is a similar algorithm to binary search. Using the pseudocode below, implement and test a function that takes a sorted list and a value as input and performs a galloping search. In general, binary search performs better than galloping. Can you think of cases where galloping search would be faster?

GallopingSearch(list, searchvalue):

1. Set start to 0 and end to 1
2. While end < length of list and the value at index end in the list < search value:
    2a. Set start to end
    2b. Multiply end by 2
3. Use another search (e.g., linear, binary) to look between the final start/end values

Note: You can modify the binary search function from the lecture to accept start/end value inputs (instead of initializing them to 0 and len(list)-1) and use that function to search between the start/end values identified by galloping search.

Problem 2

Dictionaries are very nice because they let us insert, remove and determine if an item is present in O(1) time (i.e., the time does not depend on the number of items in the dictionary). They do not, however, allow us to maintain the order that the items are added, which may be important in certain applications. 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 a solution 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, download the listandhash.py file from the tutorial page. This file has add and remove functions that modify a list variable similar to how a queue works. This 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(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. Try solving this problem yourself, but if you need additional hints, look at the end of the tutorial document or ask a TA.

Problem 3

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 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).

Problem 4

Write a recursive function called multiply(int,int) which calculates the value of a*b without using multiplication (only use addition/subtraction operators for calculations). To do this, think of how multiplication is really computed in the most basic sense and identify the base/recursive cases. Test out the function with various numbers.

Problem 5

A previous tutorial involved converting a string to Morse code. The iterative (i.e., non-recursive) solution is included in the recursivemorse.py file that you can download from the tutorial page. Add an additional recursive function that is capable of converting a given string argument to Morse. You can test your recursive function by comparing its output to that of the provided iterative solution.

Problem 3 – Additional Information/Hints

Hint #1: Note that answering the question ‘does value x exist in the list?’ is equivalent to answering the question ‘is the frequency of x in the list greater than 0?’.

Hint #2: To accomplish the goal of the problem, then, you can add a dictionary that stores the frequency of each item inside the list. In this case, the values stored in the list are the keys of the dictionary and the values of the dictionary are the frequencies of those key values in the list.

Hint #3: The frequency of an item only changes during insertion/removal, so you just need to update the item frequencies in the dictionary inside of these functions and add the improved containshash(value) function.

Hint #4: The containshash function can look up the frequency value in the dictionary. Assuming the counts are updated correctly on insertion/removal, if an item exists in the dictionary and has a count greater than 0, the item is in the list. This is an O(1) time operation.