COMP 1405B
Fall 2019 – Practice Problems #8

Objectives

- Continue to think about runtime and space complexity of algorithms
- Implement search and sort algorithms

Remember to test your solutions to verify their correctness and discuss your solutions with both other students and TAs in the course. Verifying your solutions with TAs will be a good way of ensuring that you are on a path to success in the course.

Problem 1 (Galloping Search - *)

'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., 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. What would the worst-case runtime complexity of this galloping search algorithm be?

Problem 2 (Counting Sort - **)

In general, the best performance you can hope to accomplish with sorting is an O(n log n) solution. When you know that the number of unique values you will be sorting is relatively small compared to the size of the list (n), you can improve on this solution by using a method called counting sort. The general premise of counting sort is as follows:

1. Iterate over the list one time and count the frequency of each value that occurs. This can be done in O(n) time.

2. Sort the list of unique values, which is much smaller than the size of the list, and therefore much faster to sort. This can be done in O(k log k) time, where k is the number of unique values in the list. Since k is much less than n, this is much faster than O(n log n)
3. Generate a new, sorted list by iteratively adding the correct number of repetitions of each unique value in the correct order. This can be done in O(n) time.

Create a Python file called **count.py** and implement a function called **countsort** that takes a list as input. This function must return a new, sorted copy of the input list using the counting sort method described above. Note – for this question, you can use a built-in sorting function that Python offers to sort the unique values or implement any other sorting algorithm if you prefer.

Problem 3 (Quicksort - ***)

For this problem you must implement the recursive quicksort algorithm covered in lecture to sort points based on their distance from the point (0, 0). Crete a Python file called **quicksort.py** and implement a function called **quicksort(list)** that does the following:

1. Takes a 2D list argument that contains information representing x/y points in 2D space. Each element in the list will be a list with 2 items – an x and a y coordinate. For example, the list could be [[0, 1], [2, 1], [3, 3], [1, 1], …]
2. Performs an in-place sort (i.e., does not create a new list, but modifies the original) using the quicksort algorithm. This must sort the 2D list such that points with lower Euclidean distance to the origin (0, 0) appear earlier in the list. In this case, you are comparing distances instead of directly comparing list values – it may be useful to implement and use a distance calculation function. Note – the Euclidean distance of a point (x, y) from the origin (0, 0) can be calculated with the following equation:

$$\text{distance}(x,y) = \sqrt{x^2 + y^2}$$

3. Does not return a value. As the input list is sorted in place, it will be modified directly and these modifications will be reflected outside the function, so a return value is not needed.

While the first function that will be called is quicksort(somelist), you will likely want to create helper functions to simplify your code.

Example outputs of original lists and the same lists after the quicksort function has been called:

List before sorting: [[2, 1], [2, 2], [3, 3], [3, 1], [1, 1], [1, 2]]
List after sorting: [[1, 1], [1, 2], [2, 1], [2, 2], [3, 1], [3, 3]]

List before sorting: [[3, 3], [2, 2], [1, 2], [2, 1], [3, 1], [1, 1]]
List after sorting: [[1, 1], [2, 1], [1, 2], [2, 2], [3, 1], [3, 3]]
List before sorting: [[1, 1], [3, 3], [2, 1], [2, 2], [1, 2], [3, 1]]
List after sorting: [[1, 1], [1, 2], [2, 1], [2, 2], [3, 1], [3, 3]]

Problem 4 (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 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 (Comb Sort - ***)

Comb sort is a variation of bubble sort that generally performs more efficient sorting. It accomplishes this by moving low values near the end of the list further toward the front of the list than bubble sort would during early iterations. Pseudocode for the comb sort algorithm is included at the end of this problem. Create a Python file called **comb.py** and implement a function called **combsort(list)** that does the following:

1. Takes a 2D list argument that contains information representing x/y points in 2D space. Each element in the list will be a list with 2 items – an x and a y coordinate. For example, the list could be [[0, 1], [2, 1], [3, 3], [1, 1], …]
2. Performs an in-place sort (i.e., does not create a new list, but modifies the original) using the comb sort algorithm. This must sort the 2D list such that

points with lower Euclidean distance to the origin (0, 0) appear earlier in the list. In this case, you are comparing distances instead of directly comparing list values – it may be useful to implement and use a distance calculation function. Note – the Euclidean distance of a point (x, y) from the origin (0, 0) can be calculated with the following equation:

$$\text{distance}(x,y) = \sqrt{x^2 + y^2}$$

3. Does not return a value. As the input list is sorted in place, it will be modified directly and these modifications will be reflected outside the function, so a return value is not needed.

Example outputs of original lists and the same lists after the combsort function has been called:

List before sorting: [[2, 1], [2, 2], [3, 3], [3, 1], [1, 1], [1, 2]]
List after sorting: [[1, 1], [1, 2], [2, 1], [2, 2], [3, 1], [3, 3]]
List before sorting: [[3, 3], [2, 2], [1, 2], [2, 1], [3, 1], [1, 1]]
List after sorting: [[1, 1], [2, 1], [1, 2], [2, 2], [3, 1], [3, 3]]
List before sorting: [[1, 1], [3, 3], [2, 1], [2, 2], [1, 2], [3, 1]]
List after sorting: [[1, 1], [1, 2], [2, 1], [2, 2], [3, 1], [3, 3]]

Comb Sort Pseudocode

You can use the following pseudocode as a starting point for your solution.

```
combsort(input):
    Set gap to be the length of the list
    Set shrink to 1.3
    Set sorted to False

    while sorted is False:
        Set gap to floor(gap / shrink) (i.e., round down or convert to integer)
        if gap > 1:
            Set sorted to false
        else:
            Set gap to 1
            Set sorted to true

        Set i to 0
        while i + gap < length of list:
            if input[i] > input[i+gap]:
                swap input[i] and input[i+gap]
                Set sorted to false
            Set i to i + 1
```