Lab 2: Sorting, Input/Output, Object-Oriented Programming

1 Objectives

- **Understanding Sorting Algorithms**
- **Learning how to read and write data**
- **Becoming familiar with Object-Oriented Programming**

2 Resources

- **Classes:** `https://www.youtube.com/watch?v=trOZBgZ8F_c`
- **Object Oriented Programming:**
  `https://www.youtube.com/watch?v=pxbdnrjf-Uc&list=PL82YdDfxhWsAyY3iNNDC1kUKWAJibUGi6`

3 Preliminaries

3.1 Random Array Generator

Define a function that takes as input an integer n and outputs a random array of size n. The range of values for the array can be from 0 to n. Hint: you can use the *random* module. `https://docs.python.org/3/library/random.html`**.**

4 Linear Search

In a linear search, we start from the beginning of the array and search for the value. Take your random array function and write a program that will perform a linear search for some value k on an array of size 10, 100, 500, 1000, and 5000. The program will output the position in the array that k was found, as well as the total execution time. If k is not in the array, then output "k is not found" as well as the execution time. (Hint: you can use the time module and time.time() function to track. Note that this tracks real time, so if the computer is busy with other tasks, the time output could be longer than the true execution time.)

5 Binary Search

Remember in class we discussed the pseudocode for finding someone in the phone book by dividing the book in half. Write a program that will create a sorted array of size n (with unique values). So if n = 3, the array would be [0,1,2]. If n = 10, the array would be [0,1,2,3,4,5,6,7,8,9]. Now perform a binary search for some value k. Output the index of the array in which k is located, as well as the time it takes to find k. Do this for an array of size 10, 100, 500, 1000, and 5000.

6 Homework Questions

You may not use any additional libraries besides time and random.

6.1 Sorting (30 points)

1. **In lab2.py, write functions for bubble sort, insertion sort, selection sort, and merge sort. You do not need to create the code yourself. You can use any online source as long as you cite it. [We will be comparing sorting times, so make sure that you understand the exact input and outputs of the sorting algorithms so that they are consistent.] For each algorithm, include comments explaining the code.**

2. **We want to compare execution times between all four sorting algorithms. Use the same randomly generated array on each algorithm for arrays of size 8, 200, 500, 1000, and 10000. For each array size, run the program 100 times. (Hint 1: You can use the time module to track execution time, and the write() method to output times for each iteration to a file.) Present your results in a table. Explain the results. (Hint 2: You may find it useful to run this either overnight or when you are doing something else as the larger arrays may take some time to finish.)**

6.2 Read and Write (10 points)

In the function "readFile()" in lab2.py, import the file "testNumbers.txt" and use merge sort to sort all the numbers in ascending order (i.e., smallest to largest). Write this to a new file called "SortedNumbers.txt" with the label "Sorted Numbers" as the first line, and each number on its own separate line. Your function should be flexible enough that it can account for any size list. In other words, it should work whether there are 10 numbers or 10000 numbers to be sorted.

6.3 Vigenere Cipher (25 points)

The Caesar Cipher is one of the oldest ways of disguising writing, and is attributed to Julius Caesar who used it in his private correspondences. The cipher simply shifts each character by a fixed number of positions down the alphabet. For example, a shift of 5 turns A into F, and Y goes all the way back around to D. The secrecy of this system relied on only Caesar and the recipients knowing the number of places by which Caesar had shifted his letters. This isn't particularly secure as you can just iterate through all the possible shifts to decode the message. The original message is called *plaintext*, the encrypted message is called *ciphertext*, and the secret is called a *key*.

Vigenere's Cipher improves upon Caesar by encrypting messages using a sequence of keys (or, put another way, a keyword). In other words, if p is some plaintext and k is a keyword (i.e., an alphabetical string, where A represents 0, B represents 1, C represents 2, ..., and Z represents 25), then each letter c_i in the ciphertext c is computed as: $c_i = (p_i + k_j)\ mod\ 26$.

Note this cipher's use of k_j as opposed to just k. If k is shorter than p, then the letters in k must be reused cyclically as many times as it takes to encrypt p.

In other words, if Vigenere himself wanted to say HELLO to someone confidentially, instead of using a keyword of ABC, he would encrypt the H with a key of 0 (i.e., A), the E with a key of 1 (i.e., B), and the first L with a key of 2 (i.e., C), at which point he'd be out of letters in the keyword and so he'd start the keyword again. The second L encrypts with a key of 0 (i.e., A), and the O encrypts with a key of 1 (i.e., B). So, he'd write HELLO as HFNLP.

1. **Write a program in "vigenere" in lab2.py that prompts the user for a string of plaintext, then prompts the user for a keyword, and then outputs to the screen the encrypted text. You may assume that the keyword is composed entirely of alphabetical characters. You must treat A and a as 0, B and b as 1, ..., and Z and z as 25. Your program will only apply the cipher to a character in p if that character is a letter. All other characters (numbers, symbols, spaces, punctuation marks, etc.) must be outputted unchanged. Moreover, if your code is about to apply the j^{th} character of k to the i^{th} character of p but the latter is a non-alphabetical character, you must wait to apply that j^{th} character of k to the next alphabetical character in p; you must not yet advance to the next character in k.**

2. **CHALLENGE: Codes are easy to crack if you have the keyword. But what happens if you don't know what the keyword is? You can try to brute-force a solution by using a dictionary attack, where you try using every single word in a dictionary to find the key. Now assume you are given two files: a dictionary of words in validwords.txt and cipher.txt. The keyword is one of the words in validwords.txt. Your job is to write a program to decrypt the Vigenere cipher. Your program must read in the cipher, and then cycle through the list of words until you find the right keyword. Then, your program will output the original plaintext message. Implement this in the function decrypt() in lab2.py.**

6.4 Tortoise and the Hare (25 points)

1. **Create an Animal class with 'name' and 'color' as properties in the constructor method, and a 'speak' method that returns a 'hello' message.**

2. **Python inheritance lets us define a class that inherits all the methods and properties from another class. The parent class is called the base class, and the child class is the class that inherits from another. See `https://www.w3schools.com/python/python_inheritance.asp`. Create a child class called Runner that inherits from the Animal class. In addition, Runner has an additional property of 'speed' that can be set by the user.**

3. **You will be creating a race between two instances of Runner. One is a Tortoise, and one is a Hare. The race will be visualized in the terminal, and the winner will be determined based on the speed property. You can create the race and visualization however you like, but here's some example code that may be helpful in getting you started:**

```
from random import randint
import time


def clear():
    print("\n"*80)

def race(mice):

    mice = ['----{,_,">' for i in range(mice)]
    finished = [False,]

    while not all(finished):

        time.sleep(1)
        clear()

        for i in range(len(mice)):

            x = randint(1, 6)
            spaces = ' ' * x
            mice[i] = spaces + mice[i]
            print(mice[i])

        finished = map(lambda mouse: False if len(mouse) < 50 else True, mice)

race(2)
```

7 Submitting Homework to ELMS

Make sure to list all assumptions, and use comments to notate your code whenever possible. Clarity and style will matter. Your TF should be able to read your code and understand everything. You MAY NOT use additional libraries other than random or time. Do not use any built-in sorting functions. You will be submitting two files.

1. **A python .py file that contains all comments and pseudocode**

2. **An pdf file that contains your explanations and write-ups.**

Do not change the name of the files. Add your name and section number to the top of each file as a comment.

8 Reference

- **Python commands cheat sheet** `https://ehmatthes.github.io/pcc/cheatsheets/README.html`
- **Time Library** `https://docs.python.org/2/library/time.html`
- **Classes** `https://docs.python.org/2/tutorial/classes.html`