Archie Ross
2/8/23

CSE3500 Homework 2

Question 1: **Merge Sorted Lists**

Recall in class I described how one can merge two sorted lists into a single sorted list. Now suppose you are given k sorted lists with n elements in total, and you want to merge these k sorted lists into a single sorted list. Give an O($nlogk$) algorithm for this problem and prove the correctness of your algorithm. Hint: use a heap for k-way merging.

For this question I created a python script that works as a min heap function using the heapq library. In my code I explained most of what was going on through the comments and I included the file as an attachment to this submission. Additionally I include figure 1.1 that contains a screenshot of the code alongside a terminal test that ran a simple test case to make sure the code runs as expected. I didn't run multiple test cases or edge cases as the preface of this assignment is to focus on running time not coding ability.

The time complexity of my function "merge_sorted()" can be determined as follows:

1. Biglist = k lists has a time complexity of O(n) since n is the total number of elements in all the lists combined
2. heapq.heapify(biglist) is again a time complexity of O(n), this built in command takes each element in the list and convert it into a heap
3. New_list = [] has a time complexity of O(1), it's just making an empty list
4. While biglist: is where the important part begins. This has a time complexity of O(k log k) because in each iteration of the loop the function heappop will pop the smallest element in the heap taking O(log k) time. Appending that item to the new list is only O(1), however the loop runs k times which makes the total time complexity O(n log n).
5. The return statement is a O(1) running time
6. The total time complexity can be written out as O(n) + O(n) + O(1) + O(k log k) + O(1) which can be written simply as O(n log k).

Question 2: **Run-time Analysis**

You need to implement two algorithms: Brute-force: You simply try all possible pairs of numbers to see if a given number matches each sum. Binary-search: You would first sort the list of numbers and then perform binary search on the numbers. You can use built-in sorting function. If you write sorting yourself, make sure to write an efficient sorting algorithm (O(nlogn) time).

DISCLAIMER When running my code, running the 1,000,000 numbers file against all of its targets for the brute force function simply took too long and would have prevented me from being able to submit the assignment on time. I understand that it is my fault for leaving the assignment so close to the deadline however I don't think the results of this file are necessary for me to draw a truthful and backed conclusion. This means that the table and graph lack the results of this file for this function; however it's generally understood that the runtime for the brute force function will be outrageously long. I hope this can convince you to not take points off for this…

Settings: In testing my algorithm, I did it on my AMD Ryzen 7 2700 powered personal computer with 16GBs of memory running at 2933MHz speed. The core is at stock clock with a base clock of 3.2GHz and a boost clock of up to 4.1GHz. Finally, I ran the code in VSCode which should not have affected the run time.

Results:

Table:

	listNumbers-10.txt	listNumbers-100.txt	listNumbers-1000.txt	listNumbers-10000.txt	listNumbers-100000.txt	listNumbers-1000000.txt
Brute Force Function	1.79E-05	1.17E-03	0.09393602	7.00652095	9721.213413	N/A
Binary Search Function	2.10E-05	3.26E-04	0.00409528	0.040205875	0.1331208017	1.022051108

Figure 2.1: The table representing the run time in seconds of each algorithm after running each file.

Graph:

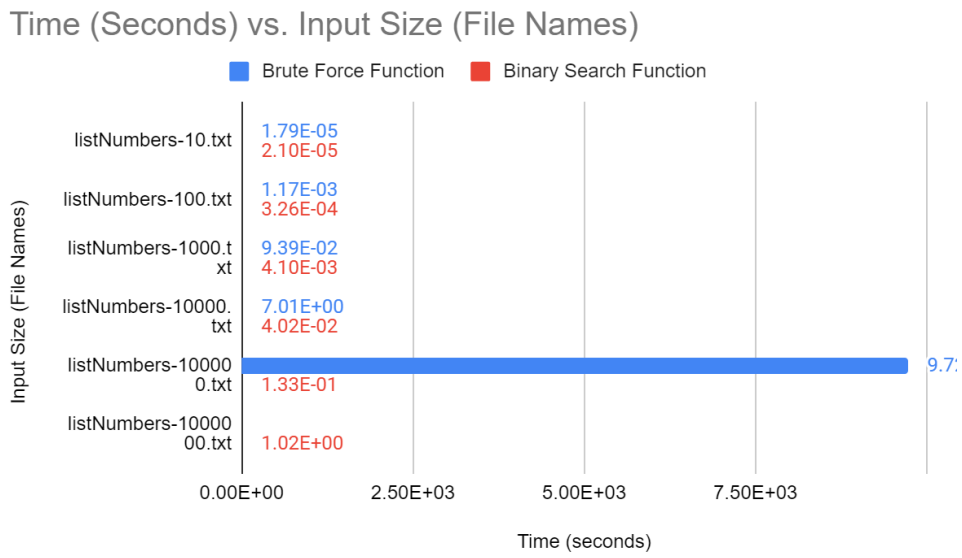

Figure 2.2: A graph expressing time vs input size of each algorithm. (There is no way for me to format the graph that makes each bar large enough to see so I added clear data points for it to make sense) Red is the Binary Search function and Blue is the Brute Force Function.

Conclusion: I think based on the running time it took to get the statistics, the table results, and the graph, it's easy to say that choosing an algorithm is significant especially when working with files or projects of high magnitude. A processor can only do so much at once and when doing things in a slow method, you're wasting time and computing power on pointless operations and instructions.

Code: *See attached to submission file Homework2 Q2.py*