CS350 Homework 4

Due 11/12 @ 11:59

1. For each of the below problems indicate which sorting algorithm would be best and why.

   Radix Sort | Bucket Sort | Counting Sort
   Quicksort | Mergesort | Insertion Sort

   (a) Analyzing 50 years of weather data collected from 1000 different stations.
   I am going to guess that we want to keep the data from each station together. Merge sort is stable which makes it best here. First sort by temperature, then by station. This will keep all the items from a station together.

   (b) Sorting the midterm exams by last name in CS350.
   Due to the very small n, the sorting algorithm efficiency doesn't matter. Simplicity is best. This makes Insertion sort the best option.

   (c) The IRS sorting the nation's tax documents using social security numbers as identifiers.
   Due to the limited set of digits and options for those digits, radix sort would be a good choice. It does well so long as d and k are limited, with d being the number of digits and k being the number of options. It also does well when all the numbers have the same d, which social security numbers do.

   (d) Sorting batting averages from historical baseball data.
   The order by year shouldn't matter, but the n will be very large. Quick sort is best in this situation due to it being fast and in place. It depends how much memory we have, but merge sort wouldn't be a bad choice either.

   (e) Sorting an array of 1,000,000 random 8-byte floating point numbers.
   Bucket sort is best here. It is a good choice for sorting floating point numbers so long as not very many options are placed into each bucket. 8 bytes is a lot of choices and so the likelihood of many collisions is low. Due to a large amount of random numbers, they should be fairly spread out.

2. Give an $O(kn \log k)$-time algorithm that merges k sorted lists each of size n into a single sorted lists. Provide pseudocode for your algorithm, a brief description of how it works, and an analysis of the asymptotic complexity.
Ex $k = 3, n = 3$
$[1, 2, 3]$
$[4, 7, 9]$
$[3, 6, 10]$
$\Rightarrow [1, 2, 3, 3, 4, 6, 7, 9, 10]$

MERGE(A(k*n), k sorted lists)

```
1   copy k_0 to B[n * k]
2   while i < k
3       Merge(A, B, k_i)
4       B = A
5       A = [n * k]
6
7   Merge(A[0...k * n − 1], B[0...k * i − 1], C[0...k − 1])
8       i = 0; j = 0; k = 0
9       Whilei < len(B)andj < len(B)
10          ifB[i] < C[j]
11              A[k] = B[i]
12              i = i + 1
13          else
14              A[k] = C[j]
15              j = j + 1
16          k = k + 1
17      ifi == len(B)
18          copy the rest of C to A
19      else
20      copy the rest of B to A
```

We already know that merge sort runs in Nlog(N) time. In this case the total number of items we need to sort is the number of lists multiplied by the number of items in each list, so nklog(N) times. The splitting takes log(N) time if we split N items in half until we reach one item. However, because we know that the lists are already sorted, we do not have to split in half all the way down. Instead we only have to split the total list into k parts. Thus the splitting only takes log(k) time. The final asymptotic complexity is then nklog(k) time.

3. A*

In order to test A * I used Spokane as a starting location and Portland as an ending location. These cities had two useful properties. First, they are close enough together that calculating the shortest route by hand had minimal error potential.

The correct route is Spokane->Vancouver_WA->Portland

Second, Seattle is closer to Spokane than Vancouver_WA but in the wrong direction. This will create a difference between the regular Dijkstra's algorithm and A*.

Dijkstra's

```
tml4@ada:~/CS350/Astar$ python3 -O graph.py
Please enter the starting location.
Spokane
Please enter the ending location.
Portland
Spokane
{'Seattle': 278.9, 'Vancouver_WA': 355.2, 'Chicago': 1785.5}
Seattle
{'Vancouver_BC': 142.6, 'Vancouver_WA': 165.1, 'Spokane': 278.9}
Vancouver_WA
{'Seattle': 165.1, 'Portland': 8.9, 'Spokane': 355.2}
Spokane ->Vancouver_WA ->Portland
The total distance was 364.09999999999997 miles.
tml4@ada:~/CS350/Astar$
```

Figure 1:

As expected, Dijkstra's algorithm investigates Seattle before finding the right path.

A*

```
Please enter the starting location.
Spokane
Please enter the ending location.
Portland
Spokane
{'Seattle': 278.9, 'Vancouver_WA': 355.2, 'Chicago': 1785.5}
Vancouver_WA
{'Seattle': 165.1, 'Portland': 8.9, 'Spokane': 355.2}
Spokane ->Vancouver_WA ->Portland
The total distance was 364.09999999999997 miles.
tml4@ada:~/CS350/Astar$
```

Figure 2:

A*'s heuristic allowed it to take the right path sooner and skip over Seattle.

The algorithm works consistently without and with the heuristic and finds the correct path. The heuristic works as intended and helps to limit the number of checks we make in order to find the correct path. The distance calculation is also correct.