lab01

September 10, 2019

1 Lab 01: Algorithm Design and Analysis

ISC 4221 Due September 11th, 2019 Connor Poetzinger

1.1 Introduction

This lab introduces brute force algorithms. I implemented two sorting algorithms, selection sort and bubble sort, to sort a real one-dimensional array in ascending order. I then apply the selection sorting algorithm to a problem to determine the closest store location from a hypothetical caller's latitude and longitude location. The stores distance are calculated via the Haversine formula and the resulting list of stores, city, and distance are then sorted by distance in ascending order.

1.2 1. Selection Sort

This algorithm takes in an array of n real numbers to be sorted and outputs the sorted array in ascending order as well as its positional index vector. The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. The algorithm proceeds by finding the smallest element in the unsorted sublist, swapping it with the leftmost unsorted element, and moving the sublist boundaries one element to the right.

```
In [1]: #Import modules
        import numpy as np
        """
        Input: Numpy array of random real numbers from 0 to 100
        Output: Sorted array and index vector

        Goal: Take current element and swap it with the smallest element on its right
        """

        def selectionSort(A):
            #use numpy argsort to return indicies that would sort the array
            indx = np.argsort(A)
            #Traverse through numpy array
            for i in range(len(A)):
                #initial minimum location
                min_loc = i
                #find location of smallest element on right
```

```
            for j in range(i + 1, len(A)):
                #Check if number to the right is smaller then minimum location
                if A[j] < A[min_loc]:
                    min_loc = j
            #Within the first for loop swap the minimum location with first element
            A[min_loc], A[i] = A[i], A[min_loc]

        return A, indx
```

```
In [2]: A = np.random.rand(26) * 100
        print("Unsorted list or random floats from 0-100\n\n", A)
```

```
Unsorted list or random floats from 0-100

 [59.43923056 60.12798562 55.06670094 27.06336977 52.49438164 82.12712886
 28.85345318 88.79417293 19.39409166 19.83851213 40.27765917 24.1120189
 60.47495862 76.58320643 59.11134316 37.23466161 52.38744396 41.92181875
 39.33087457 12.24157373 58.95975058 29.540956   59.47847629 87.48173884
 63.84389335 88.37816628]
```

```
In [3]: B, indxd = selectionSort(A)
        print("Sorted list of random floats from 0-100 and its position indicies\n\n",
              B, "\n\n", indxd)
```

```
Sorted list of random floats from 0-100 and its position indicies

 [12.24157373 19.39409166 19.83851213 24.1120189  27.06336977 28.85345318
 29.540956   37.23466161 39.33087457 40.27765917 41.92181875 52.38744396
 52.49438164 55.06670094 58.95975058 59.11134316 59.43923056 59.47847629
 60.12798562 60.47495862 63.84389335 76.58320643 82.12712886 87.48173884
 88.37816628 88.79417293]

 [19  8  9 11  3  6 21 15 18 10 17 16  4  2 20 14  0 22  1 12 24 13  5 23
 25  7]
```

Selection sort is noted for its simplicity, it has O(n2) time complexity.

1.3 2. Bubble Sort

This algorithm takes in an array of n real numbers to be sorted and outputs the the sorted array in ascending order as well as its positional index vector. The algorithm proceeds by repeatedly sweeping through the list, compares adjacent elements and swaps them if they are in the wrong order. The sweep through the list is repeated until the list is sorted.

```
In [4]: #Import modules
        import numpy as np
        """
```

```
    Input: Numpy array of random real numbers from 0 to 100
    Output: Sorted array and index vector

    Goal: Move left to right, compare consedcutive elements and switch them if
    they are out of order. Continue until no swaps are made through an entire
    sweep.
    """

    def bubbleSort(A):
        #use numpy argsort to return indicies that would sort the array
        indx = np.argsort(A)
        #Traverse the numpy array
        for i in range(len(A)):
            #At each sweep compare the current j with the next value
            #Use length minus 1 since we are comparing the current value
            #with the next
            for j in range((len(A) - 1) - i):
                #Swap positions if element found is greater than the next
                #element. Largest nums bubble to the back
                if A[j] > A[j + 1]:
                    #Current element moves to the back
                    A[j], A[j + 1] = A[j + 1], A[j]

        return A, indx
```

```
In [5]: A = np.random.rand(25) * 100
        print("Unsorted list or random floats from 0-100\n\n", A)
```

```
Unsorted list or random floats from 0-100

 [72.56893187 76.16464016 29.82701629 25.11089047 88.5028215  43.43356996
 49.1445192  70.33294691 38.50901398 49.55203914 34.43311064  1.69675518
 96.73588528 64.46356414 58.62491947 88.58874741 82.18190962 90.73361751
 50.07136809 65.86577621 87.28108278 31.49009725 55.43346748 52.52842143
 51.7450819 ]
```

```
In [6]: B, indxd = bubbleSort(A)
        print("Sorted list of random floats from 0-100 and its position indicies\n\n",
              B, "\n\n", indxd)
```

```
Sorted list of random floats from 0-100 and its position indicies

 [ 1.69675518 25.11089047 29.82701629 31.49009725 34.43311064 38.50901398
 43.43356996 49.1445192  49.55203914 50.07136809 51.7450819  52.52842143
 55.43346748 58.62491947 64.46356414 65.86577621 70.33294691 72.56893187
 76.16464016 82.18190962 87.28108278 88.5028215  88.58874741 90.73361751
 96.73588528]
```

```
[11  3  2 21 10  8  5  6  9 18 24 23 22 14 13 19  7  0  1 16 20  4 15 17
 12]
```

Bubble sort is noted for its simplicity however, this algorithm is noticeably slower than selection sort. Bubble sort has a complexity of O(n2) which is the same as selection sort, but since bubble sort takes multiple sweeps of the list to sort the array, it is considered inferior to selection sort.

1.4 3. A Basic Application of Sorting

Using the Haversine algorithm, find the distance between the store distance and the user inputed longitude and latitude. The example latitude and longitude I use is 82 W 29 N, approximately near Ocala, Florida.

```
In [7]: import numpy as np
        import pandas as pd

        def haversin(data):

            """
            This function reads in user latitude and logitude to simulate logging customer
            call locations. The user's long and lat are then converted to radians along
            with other pre-defined longs and lats imported from a text file and save to a
            pandas dataframe. The function then computes the distance from the caller to
            the the stores in the dataframe using the haversine formula. The distances are
            then sorted in ascending order and printed out to provide the customer with
            a list of closest stores, and how many miles to the store.
            """

            #Read in user long and lat
            #must transform string values to float
            lon1 = float(input("Enter longitude: "))
            lat1 = float(input("Enter latitude: "))

            #Radius of earth from the equator in miles (found on google)
            R = 3963.0

            #assign user lat and lon and datatable lat and long to variables
            #I use the in-built function map to assign the numpy function
            #np.radians to the degree
            #lats and long to transform degree into radians
            lat1, lon1, lat2, lon2 = map(np.radians,
                                         [lat1, lon1, data.latitude, data.longitude])

            #calculate the distance between the lats and longs
            lon_dist = lon2 - lon1
            lat_dist = lat2 - lat1
```

```python
        #apply haversine formula
        #np cos and sin provide for faster calculations
        c = 2 * R * np.arcsin(np.sqrt((np.sin(lat_dist)/2)**2 +
                                      np.cos(lat1) * np.cos(lat2) *
                                      np.sin((lon_dist)/2)**2 ))

        #prompt user for the info provided
        print("\nBelow are the closest stores from your location in ascending order\n")

        #apply selection sort
        sorted_result, indx = selectionSort(c)

        return sorted_result, indx
```

In [8]:
```python
#import data table
#use pandas to assign columns using strings
data = pd.read_table('stores_location.dat', delim_whitespace=True,
                     names = ('store', 'city', 'latitude', 'N', 'longitude','W'))

#call function
sorted_dist , indx = haversin(data)

#Sort the cities and stores based on the index from selectionSort(c)
sorted_cities = [data['city'][indx[i]] for i in range(len(data))]
sorted_stores = [data['store'][indx[i]] for i in range(len(data))]

#create a tuple to package store num, city, and dist together
sorted_zip = list(zip(sorted_stores, sorted_cities, sorted_dist))

#Create output dataframe
final_sort = pd.DataFrame(sorted_zip, columns=['Store','City','Distance(m)'])
print(final_sort.to_string(index=False))
```

```
Enter longitude: 82
Enter latitude: 29

Below are the closest stores from your location in ascending order

Store           City  Distance(m)
store#2   Gainesville    49.170962
store#6       Orlando    58.666064
store#5         Tampa    77.023368
store#4  Jacksonville    94.676263
store#1   Tallahassee   168.646953
store#7       Hialeah   241.000734
store#3         Miami   248.602665
```

Testing the longitude and latitude coordinates for Ocala, Florida (82 W, 29 N), the resulting list is accurate. The closest city, Gainesville, is 49.17 miles away followed by Orlando at 58.66 miles away. The furthest city from the given cooridinates is Miami at is 248.60 miles away.