Keiffer Tan

Insertion Sort Analysis

```python
""" Implementation of Insetion Sort """
def insertion_sort(arr):
  n = len(arr)

  for i in range(1, n):
    key_val = arr[i] # store the key value that will be inserted into the "sorted" part of the array
    j = i-1 # j will be the top index of the "sorted list" section

    # go down the "sorted list" section of the array to find where the key value will be inserted
    while j >= 0 and key_val < arr[j]:
      arr[j+1] = arr[j] # shift elements to the right of the list/array
      j -= 1
    arr[j+1] = key_val # insert key_val into correct index

  return arr
```

I have the time complexity analysis on the next page

For space complexity analysis, insertion sort is also considered an in-place sorting algorithm. The space complexity is O(1) or constant because we are shifting elements within the given array/list and inserting the key value in the correct place of the “sorted” portion of the array/list. We are not allocating a lot of new memory for this other than storing a few key variables.

```python
""" Implementation of Insetion Sort  w/ tuple input to test stability """
def insertion_sort_tuple(arr):
  n = len(arr)

  for i in range(1, n):
    key_val = arr[i] # store the key value that will be inserted into the "sorted" part of the array
    j = i-1 # j will be the top index of the "sorted list" section

    # go down the "sorted list" section of the array to find where the key value will be inserted
    while j >= 0 and key_val[0] < arr[j][0]:
      arr[j+1] = arr[j] # shift elements to the right of the list/array
      j -= 1
    arr[j+1] = key_val # insert key_val into correct index

  return arr
```

For stability, insertion sort is a stable sorting algorithm and preserves the relative order of same value items. In my code above, I altered the original insertion sort algorithm to accept an array of tuples and only compare the first numerical value so that we can observe the 2nd char value in the tuple.

```python
arr = [(2, 'b'), (2, 'a'), (1, 'c'), (1, 'a')]
expected = [(1, 'c'), (1, 'a'), (2, 'b'), (2, 'a')]
```

In my tests, I input the arr into the algorithm and I assert that the output is equal to the expected array. Since all the tests pass, I assume that the relative order of same value is preserved as (2,‘b’) is before (2,’a’) in the original and expect arrays.

```
Time taken by Empty Array: 0.000003500 seconds
Time taken by Single Element Array: 0.000001000 seconds
.Time taken by Randomly generated array: 0.000006900 seconds
.Time taken by Same Elements Array: 0.000001300 seconds
.Time taken by Descending Sorted Array: 0.000009800 seconds
.Time taken by Sorted Array: 0.000001300 seconds
.Time taken by Tuple Array1: 0.000002800 seconds
Time taken by Tuple Array2: 0.000001400 seconds
Time taken by Tuple Array3: 0.000000700 seconds
```

I also have my testing time recorded as well. Looking at the best case scenario time it took to sort a sorted array is 1.3 microseconds and the worst case scenario of an opposite sorted array took 9.8 microseconds. I would say these test numbers do match the worst, average, and best time scenarios of the sort.

- The best-case scenario performs well in already sorted or almost-sorted arrays because it can traverse through the array n times since it will not need to insert any elements into the "sorted" portion of the array.
- The worst-case scenario performs terribly in reverse sorted order because the algorithm will have to insert every single item into the "sorted" portion of the array for every iteration. This will cause an n*n time with the worst case
- Even though theoretically average time is O(n^2), the time I have from randomly generated arrays seems to be somewhere between 5.5 – 7 microseconds when I run multiple tests. Though if I tracked every single run for a large sample size, I will probably find out that it will be O(n^2) since I only did a small sample size, it makes sense my findings are slightly skewed.
- `Time taken by Large Randomly generated array: 5.103485000 seconds`
    - For a randomly generated array with 20,000 elements between 0-2000 took 5.1 seconds to completely sort with insertion sort.

Insertion sort does work very efficiently with smaller datasets since it can just insert into the "sorted" portion of the array/list quickly. Though as the list gets larger and larger, the algorithm can slow as it may have to find the correct position to insert in the list. For example, if the list is 1,000,000 elements long and it is on the 900,000 iteration and it is trying to find the position of the next key value which is the lowest value in the entire list, the algorithm will need to compare 900,000 times until it reaches the beginning before inserting the key value. This can be very cumbersome and inefficient. Compared to the Merge sort algorithm which does very well in time, it trades it off with a worse space complexity.

In practical applications, insertion sort would be best used in a library. Librarians, when organizing books tend to find the spot the book belongs and inserts it in the proper position.

For some improvements and variations, I thought of larger data sets when it is trying to find where the key value belongs in the "sorted" portion of the array/list, rather than starting from the largest value, we can try and find the position by comparing the middle index of the "sorted" portion with the key value and trying to find the correct index by further comparing pivot points. This can possibly reduce the time to find the correct index. It is kind of like finding the correct position when inserting into a binary tree, there the parent node is the pivot point of going left or

right if it is smaller or larger than the parent node. I believe the biggest drawback of insertion sort with large datasets would be finding the correct index in the “sorted” portion of the array/list.