Workbook Assignment B

Patrick Hamer

Q1. Identify and explain the workings of TWO sorting algorithms and discuss and compare their performance/efficiency (i.e. Big O)

Bubble Sort is an algorithm in which each item is compared to the next, and if out of order then they are switched. For example if the user was trying to get a low to high order from the list:

```
[5, 1, 4, 2, 3]
```

then for each index in the array the algorithm would have to go through each index in the array to see if it was larger than the one after it:

```
for i in range(n-1):
    for j in range(0, n-i-1):
        if arr[j] > arr[j + 1]
```

Thus allowing the largest numbers, one by one to slowly make their way to the end, or *bubble* their way up to the top of the array. While this may be an effective algorithm for sorting, with a best case scenario of O(n) if the list is already in order and a worst case of O(n^2) if the list is completely in reverse. The reason this would at best be an O(n) is because if the numbers were already in order, and were added in order, then the script would only have to iterate through the the amount of inputs. The quadratic notation (O(n^2)) at worst is due to the fact that for every additional element added to the array, the algorithm would not only have to iterate through that extra element itself, but that element would be added to every single iteration thus creating the extra processing time. For example, it the list was as simple as

```
[3, 1, 4, 7]
```

then 10 was added, the algorithm would now have to sort through 5 numbers as opposed to the original 4 for each iteration, as well as a whole extra 5 numbers at the end as it iterated for the added number.

Merge sort is a 'divide-and-conquer' algorithm whereby the array is borken down into smaller and smaller temp-arrays until it is simply a group of arrays each containing one element. For example:

```
[5, 9, 2, 7, 1]
[5, 9] [2, 7, 1]
[5] [9] [3] [7] [1]
```

Then the arrays are compared and merged one by one as appropriate. For example:

```
[5] [9] [3] [7] [1]
```

will be merged into

```
[3, 5, 9] [1, 7]
```

next the smallest numbers will be compared from the list and the smallest will be added back to the original:

```
[1]
[3, 5, 9] [7]
```

Now again, the two smallest numbers will be compared and the smallest will be added back to the original list:

```
[1, 3]
[5, 9] [7]
```

and so on until the list is recompiled in the correct order. This merging is somewhat more efficient as it has a time complexity of O(nlogn) due to the fact that it does not require an iteration of every single element for every single index of an array. That is to say that whereas in the bubble sorrting of the array

```
[5,4,3,2,1]
```

The 5 will have to compared first to the 4, then the three, then the 2 then the 1 until it bubbles to the end of the list, in merge sort it will be compared only with each element once. Example being that if you have

```
[5, 4, 3, 2, 1]
```

broken down to individual elements in arrays then the first merge will automatically put it on the other side of 4, meaninng that when the lowest bumber in each array is compared 5 will be absent from the comparison until it is reached in the reassambled array. You would end up with

```
completed_array = [1, 2, 3, 4]
[5]
```

without 5 having been compared to any number besides 4. This skips a lot of the double handling you find in the bubblesort algorithm, however it does cost more *space* to do. The Algorithm has a complexity of O(n) for space complexity as for each extra element added to the array another split has to be performed to get down to the single-number arrays required before merging.

This means that Merge Sort would be more efficient for longer lists, as you would lose all that double handling and reiteration however for shorter lists it may not be worth the extra space that it takes up, and also if a list is tiny, say like the examples we used, then bubble sort

could actually be quicker due to the fact that it only needs to bump up a number a few places as opposed to splitting an entire list and reassembling it.

Identify and explain the workings of TWO search algorithms and discuss and compare their performance/efficiency (i.e. Big O)

Linear Search is an algorithm that checks each item in a list in sequence to see if it is the desired result. For example in the list

```
array1 = [4, 10, 15, 6, 9, 0]
```

to find 6 the algorithm would work its way through each index checking. For example:

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

until it found the index that was the same as x (in this case, 6). This algorithm will continue searching for x until it either finds it or reaches the end of the data set, in whicvh case it will return a -1 and end. This form of searching is simple and effective, however for larger data sets not so efficient. It has a complexity of O(n) whereby as the data input grows so does the complexity of the algorithm as it has more and more data to work through.

in **Binary Search** however, we once again see the efficiency of the divide and conquer approach. To search for x in Binary Search, the array must be ordered, then the algorithm will: Locate the mid point. If the mid point == x, return the index. If the mid point < x then the left side of the array will be ignored and a midpoint between the original mid and the end of the array will be found and the process will be repeated until x is found. Example:

```
while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
                found = True
        else:
                if item < item_list[mid]:
                        last = mid - 1
                else:
                        first = mid + 1
return found
```

(https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-1.php, 2022)

This search method has a complexity of O(nlogn) as the more times it repeats itself the more of the data will be ignored, which means that while initially if you have large quantities of data they will soon become small and quite managable. To compare the two, imagine you

have 1000 elements in an array, and x is to be found at the 900th element. This means that for a linear search there would have to be 900 iterations of the algorithm until x was found. In binary search however, there would be 10 at most as each recursion would be focusing on a small and smaller selection of input. The downside of binary of course, is that the data needs to be in order in the first place which, if it is not, could be a time consuming process in and of itself.

References:

While I have attributed the code I used directly from a website, the majority of what I have learned about these subjects is from general perusing and discussion with friends/educators who are in the industry. Some resources I found myself consistently using were:

- https://www.w3schools.com/
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/