Assignment 2

Nicholas Maisel

NOTE: The sortingAlgorithms file imports random from random.

1 Insertion Sort

Listing 1: Insertion Sort Source Code

```
1  insertionComparisons = 0
2  def insertionSort(A):
3      global insertionComparisons # Comparison global variable access
4      for i in range(0,len(A)):
5          key = A[i]
6          j = i-1
7          while j>=0 and key < A[j]: # Waits until key is greater than A[j]
8              insertionComparisons +=1
9              A[j+1] = A[j]
10             j -= 1
11             A[j+1] = key # Inserts the value
12     return(A)
```

1.1 Results

Sort	Comparisons	$O(n)$
Insertion Sort	114317	$O(n^2)$

The Insertion Sort Algorithm runs at $O(n^2)$. The reason that it is n^2 is because of the way the for-loop on line 4 works. We are iterating n times in the for-loop, because it goes from the first element to the very last element, which is found using 'len(A)'. For each increment of i, we search through n-i elements of the list in our while-loop on line 7. Therefore we are iterating in the following pattern:

(n-1) + (n-2)...(n-(n-1))

Which can be shown as:

$(n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{n(n-1)}{2} \sim \frac{n^2}{2} = O(n^2)$

The reason that we can go from $\frac{n(n-1)}{2} \sim \frac{n^2}{2} = O(n^2)$ is because as n approaches ∞ the constant becomes irrelevant. Our Algorithm is sorting a list of magicitems with a length of 666. $666^2 = 443556$, which might appear to be wrong however, remember the constant we threw away? That is the reason that our number of comparisons was 114317 rather than 442556.

2 Selection Sort

Listing 2: Selection Sort Source Code

```
1  selectionComparisons = 0
2  def swap(Arr, j, i):
3      tempVar = Arr[j]
4      Arr[j] = Arr[i]
5      Arr[i] = tempVar
6      return(Arr)
7
8  def selectionSort(A):
9      global selectionComparisons # Used to count the number of comparisons
10     n = len(A)
11     for q in range(0, n-1):
12         smallPos = q
13         for x in range(q+1, n):
14             # Increments comparisons each time one is done, it is before
15             # the inequality, because the inequality may not always be true.
16             selectionComparisons += 1
17             if A[x] < A[smallPos]:
18                 smallPos = x
19         A = swap(A,q,smallPos)  # Calls the swap function
20     return(A) # Returns tuple of the sorted list and # of comps
```

2.1 Results

Sort	Comparisons	$O(n)$
Selection Sort	221445	$O(n^2)$

The Selection sort algorithm runs at $O(n^2)$. The reason the algorithm runs at n^2 is because of the nested for-loops on lines 12 and 14. The first for loop loops from 0 to n-1. Therefore we are already running at $O(n)$ (Again, drop the constant). But we still have to iterate through out next loop, which ranges from q (the value provided by the first loop) to n. This relation can be shown as:

$$(n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Our algorithm is again sorting the list of magic items that has a length of 666. $666^2 = 443556$. But our algorithm returned 221445 comparisons. This is because as n approaches infinity constants become irrelevant. I found this result to be especially interesting, it is almost exactly $\frac{1}{2}$ of (n^2).

3 Merge Sort

Listing 3: Merge Sort Source Code

```python
mergeComparisons = 0
def merge(left, right):
    global mergeComparisons
    sortedList=[] # Array used to store the sorted list
    i,j=0,0
    while i<len(left) and j<len(right): # Ensures we dont go out of list
        mergeComparisons +=1
        if left[i] < right[j]:          # Comparing the unitary lists
            sortedList.append(left[i])
            i+=1                        # Moves to next element in list
        else:
            sortedList.append(right[j])
            j+=1

    sortedList+=left[i:]
    sortedList+=right[j:]
    return (sortedList)

def mergeSort(A):
    n = len(A)
    if (n <= 1):
        return(A)
    splitPoint = int(n/2)
    # Defines left as a list with everything up to the splitpoint
    left = mergeSort(A[:splitPoint])
    right = mergeSort(A[splitPoint:])

    return(merge(left,right))
```

3.1 Results

Sort	Comparisons	$O(n)$
Merge sort	2989	$O(n^2)$

The Merge sort algorithm runs at $O(n \log_2 n)$. The reason the algorithm runs at $O(n \log_2 n)$, is easier to explain in two parts. We first have to split our list into individual items. We do so by breaking the list into halves each time (If n is odd, we break it into an individual item and a list with an even length). But, how many splits do we have to perform until we have completely broke down the list? We can use $(\log_2 n)$ to find this. Now where does the n come into the equation? The n comes from the merging part of the algorithm. We merge n items at each pass, meaning we will only need a max on n comparisons to do so, Thus $(n \log_2 n)$

4 Quick Sort

Listing 4: Quick Sort Source Code

```
1   global quickComparisons
2   left = []
3   pivotList = []
4   right = []
5   if (len(A) <= 1):
6       return(A)
7   else:
8       pivotPoint = choice(A)
9       # The block below iterates over the lists and
10      #chooses which partition values belong in
11      for i in A:
12          quickComparisons +=2
13          if i < pivotPoint:
14              left.append(i)
15          elif i > pivotPoint:
16              right.append(i)
17          else:
18              pivotList.append(i)
19      leftSide = quickSort(left)
20      rightSide = quickSort(right)
21      return(leftSide + pivotList + rightSide)
```

4.1 Result

Sort	Comparisons	$O(n)$
Quick Sort	14626	$O(n^2)$

The Quick sort algorithm runs at $O(n \log_2 n)$. The reason the algorithm runs at $O(n \log_2 n)$ is because of the way it splits the arrays. As seen on line 8, the array is split at a randomly chosen pivot point. You might be asking, if it is random then how can we say it is $\log_2 n$. It is $\log_2 n$, because on average the pivot point will be in the middle, thus splitting the list into two. Then each time we split it we go through the entire list as seen in the loop on line 11, intuitively we are iterating n times. Thus we get $O(n \log_2 n)$

5 Linear Search

Listing 5: Linear Search Source Code

```
linearComparisons = 0
individualLinear = 0

def linearSearch(A, target):
    global linearComparisons
    global individualLinear
    location = 0 # Location is the index of comparison
    while location < len(A): # Ensures we dont go out of array
        individualLinear +=1 # Updates counters
        linearComparisons +=1
        if A[location] == target: # Compares values to target
            flag = True
            break # Stop if we find the target
        else:
            flag = False
        location +=1 # Move onto next index
    return(flag)
```

5.1 results

Individual Linear Search Comparisons

33	262	127	585	530	218
227	252	548	159	95	662
567	98	572	250	596	273
257	323	270	601	598	441
484	335	588	95	54	36
595	557	317	537	153	490
651	622	32	13	270	452

Search	Average Comparisons	$O(n)$
linear Search	324.2857142857143	$O(n)$

The linear search algorithm runs at $O(n)$. It is quite easy to understand the reasoning behind its performance. Linear search takes an list and a target value, it iterates over each element of the list until it finds the value. At its worst, linear search will have to traverse to the very last element in the list. But on average it will find it $\frac{n}{2}$. We see this very clearly in our average comparisons. Out of the 666 magicitems, on average the linear search algorithm searched through 324.286 items, almost exactly $\frac{1}{2} * 666$.

6 Binary Search

Listing 6: Binary Search Source Code

```python
def binarySearch(A,start,stop,target):
    global binaryComparisons # Gives method access to global var
    global individualBinary
    flag = False
    midpoint = int((start + stop)/2) # Sets midpoint to middle
    binaryComparisons += 1 # Updates counters
    individualBinary += 1
    if start > stop: # Makes sure we havent exhausted search
        flag = False
    elif (A[midpoint] == target): # Compares target
        flag = True
        location = midpoint
    elif (target < A[midpoint]): # Searches lower
        binarySearch(A,start, midpoint-1, target)
    else: # Searches upper
        binarySearch(A, midpoint+1,stop,target)

    return(flag)
```

6.1 Results

Individual Binary Search Comparisons

9	8	8	9	10	6
9	8	9	9	8	7
10	7	10	3	8	10
8	8	6	10	10	10
9	10	9	10	8	9
9	9	7	10	9	8
6	10	10	8	9	10

Search	Average Comparisons	$O(n)$
Binary Search	8.595238095238095	$O(\log_2 n)$

The Binary search algorithm runs at $O(\log_2 n)$. The reason binary search runs at $(\log_2 n)$ is because of the way the clever algorithm splits the sorted list into two based on the middle and target value. It located the midpoint and checks if the target value is greater or less than the midpoint. If it is less than it throws away the top half of the list, if it is greater than, it throws away the bottom half of the list. It takes the remaining half of the list and runs another binary search on that half. This splitting is what causes the binary search algorithm to run at $(\log_2 n)$. Our average comparison count came out to be about 8.596, if we use the inverse of the $\log_2$, which is simply raising $(2^8.596 = 386.94910)$. While the result is less than our 666 items, this is to be expected.

7 Hash Table

Listing 7: Hash Table Source Code

```python
from LinkedList import LinkedList
from LinkedList import Node
import random

class HashTable():
    def __init__(self,tableLength):
        self.tableLength = tableLength
        self.table = {}

    def hash(self,key):
        #Uses python's built in hash tool to and modding it to tableLength
        return(hash(key)%self.tableLength)


    def get(self,key):
        comparisons = 1     #Accounts for the 'get' part of each comparison
        hashedKey = self.hash(key)
        flag = False

        #check to see if the key exists in the hash table
        if hashedKey in self.table.keys():
            curNode = self.table[hashedKey].firstNode   #start out at the front
            while flag == False:
                comparisons +=1
                if curNode.valueOfNode == key:
                    #if we found the key, this will break out of the while loop
                    flag = True
                elif curNode.nextNode:
                    curNode = curNode.nextNode
                else:
                    break
        return(flag,comparisons)


    def put(self,key):
        hashedKey = self.hash(key)
        if hashedKey in self.table.keys(): #Checks to see if the hash has been used
            self.table[hashedKey].AddToFront(key)
        else:
            #If there is not already a value mapped to this hash output's
            #Linked list, simply make one and set the key to the value of that
            #linked lists's firstNode value
            self.table[hashedKey] = LinkedList(key)


# The main() function is used to find the average number of comparisons
# at any timesToCheck
def main():
    b = HashTable(255)
    f = open('magicitems.txt',"r")
    magicitems = list(f)
    f.close
    magicitems = [x.strip() for x in magicitems]
    magicitems = [x.lower() for x in magicitems]
    magicitems = [x.replace('␣','') for x in magicitems]

    a = magicitems

    for i in a:
        b.put(i)

    totalComparisons =0
    timesToCheck = 100
    for i in range(timesToCheck):
        found,comparisons = b.get(random.choice(magicitems))
        totalComparisons += comparisons

    print("Average␣Comparisons:␣", totalComparisons/timesToCheck)



#This is used to run stuff only when not called via an import
#ie. if this module is being used as an import, main will not run
#however, if it is run directly, main() will be called
if __name__ == "__main__":
    main()
```

7.1 Results

Individual Binary Search Comparisons

3	3	2	2	2	3	2
3	5	5	2	2	2	2
3	2	5	2	3	5	2
3	6	2	3	4	4	4
4	3	2	2	3	4	2
6	2	5	2	2	2	7

Algorithm	Average Comparisons	$O(n)$
Hash Table	3.36	$O(n)$

The hash algorithm is a bit confusing when we talk about it having $O(n)$. to 'get()' an item from the hash table we first need to hash the key we are searching for. The hashing happens in constant time. At this point we are at the linked list that holds the value(s) that share(s) the same hash as our key. We need to traverse the linked list to find if our value is in the linked list. Intuitively each linked list in our chain is going to have different lengths. But the average length can be defined by our hash tables load:

$\frac{Numberofitems}{Hashtablesize}$

In our case the load is $\frac{666}{255}$. But since if we have the possibility that our hash function hashes all values to the same chain, we have $O(n)$, because we may have to traverse the chain with all values.