Sorting algorithms are used to rearrange the elements of a list so that the list is in strictly increasing order. Sorting algorithms are used very frequently across computer science disciplines so it is important to find the most optimal ones. I will be analysing insertion sort, merge sort, binary insertion sort, stooge sort, heap sort, bogo sort, and quick sort. For each algorithm a pseudocode implementation will be given along with a big O analysis of the function. Graphs were made by timing each sort for lists of increasing length. A strictly descending list was used for each test as it is the furthest away from a sorted list and should give the worst case run times for each algorithm.

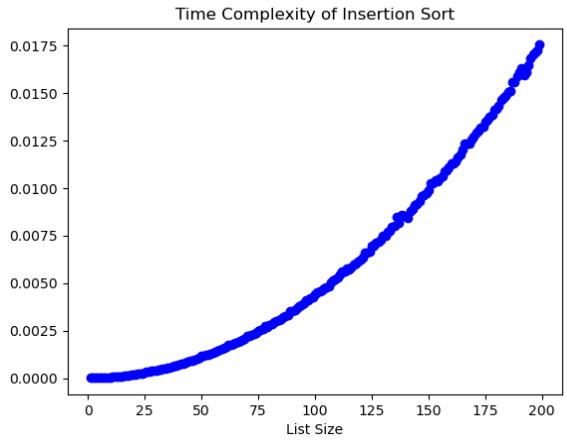
```
def insertionSort(myList):                                                    O(1)
    create a list called sortedList that is the same length as mylist\        O(1)
    with the first element being the first element in myList and the other elements being None

    for i from 1 to the length of myList:                                     O(n)
        initialize j to be i-1                                                O(1)
        while j is greater than zero :                                        O(n)
            if sorted list at j is greater than myList at i:                  O(1)
                set sortedList at j+1 to be sortedList at j                   O(1)
                subtract 1 from j                                             O(1)
            otherwise:                                                        O(1)
                break from the loop                                           O(1)

        set sortedList at j+1 to be myList at i                               O(1)

    return sortedList                                                         O(1)
```

Insertion sort works by sorting each element into a subarray on the left part of the list. This subarray is sorted and each element is added into its proper position in the subarray. The algorithm takes the element that needs to be sorted and looks at every element before it in the list. It moves every element that is larger than the one being sorted up one place in the list. Then it inserts that element into the open position.l

The overall run time is O(1) + O(1) + O(n)(O(1) + O(n)(O(1) + O(1) + O(1))) + O(1)

The majority of the run time comes from the nested loops. In the worst case run time where the initial list is strictly descending both loops will iterate roughly n times. So this insertion sort algorithm is O(n^2). Since we are using two lists of size n the memory usage will be 2n. The function does not allocate any extra memory so the big O memory requirement is O(1).

The graph shows that the runtime increases as a quadratic function of the size of the list which matches the above analysis.



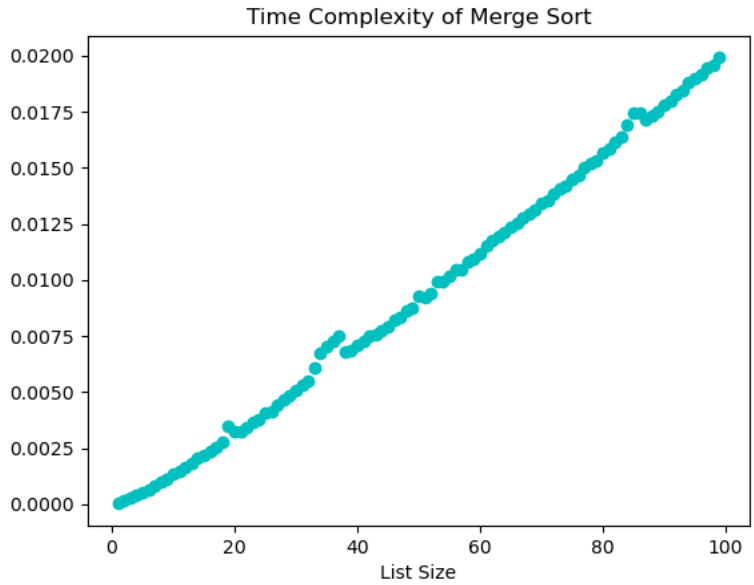

```
def mergeSort(myList):                                                          O(1)
    if the length of myList is greater than one:                                O(1)
        initialize mid to be the length of myList divided by 2                  O(1)

        initialize left to be a recursive call of mergeSort\                    T(n/2)
        with parameter myList indexed from 0 to mid
        initialize right to be a recursive call of mergeSort\                   T(n/2)
        with parameter myList indexed from mid to the end

        initialize i to 0                                                       O(1)
        initialize j to 0                                                       O(1)
        initialize k to 0                                                       O(1)

        while i is less than the length of left and j is less than the length of right:   O(n)
            if left at i is less than right at j:                               O(1)
                set myList at k to be left at i                                 O(1)
                increment i by 1                                                O(1)
                increment k by 1                                                O(1)
            else:                                                               O(1)
                set myList at k to be right at j                                O(1)
                increment j by 1                                                O(1)
                increment k by 1                                                O(1)
        while i is less than the length of left:                                O(n)
            set myList at k to be left at i                                     O(1)
            increment i by 1                                                    O(1)
            increment k by 1                                                    O(1)
        while j is less than the length of right:                               O(n)
            set myList at k to be right at j                                    O(1)
            increment j by 1                                                    O(1)
            increment k by 1                                                    O(1)

    return myList                                                               O(1)
```

Merge sort works by splitting the list in half, sorting each half, and then merging the two sorted sublists together. For each half it recursively calls mergeSort in order to create a sorted list. So we continuously split the list until we get lists of size 1, and then go back up the stack of recursive calls to merge each list. Once we get back up to the first call there we have two sorted sublists that can be merged together to create a sorted version of the original list. The algorithm for merging the two lists is as follows:

1. Compare elements in the first list to elements in the second list and place the smaller elements in the original list.
2. Once we run out of elements in one list, that means the remaining elements of the other list can be added to the merged list since they will all be larger than any current element.

Here is an analysis of the run time:
$T(n) = 2T(n/2) + n$
$a = 2$
$b = 2$
$c = 1$
$\log_b(a) = \log_2(2) = 1 = c$
So by the master theorem $T(n) = O(n\log n)$
So this merge sort algorithm is $O(n\log n)$.
Merge sort requires more memory than any of the other sorting algorithms. Since we create two new arrays that are half the size of the previous arrays each time the function recurses we create a new list that is half the size of the previous list. So the memory requirement for this algorithm is $n + n/2 + n/4 + \ldots + n/n = 2n$. So the big O memory requirement is $O(n)$

It is reasonable to assume that the graph of run times shows a run time increase that follows an nlogn growth rate.



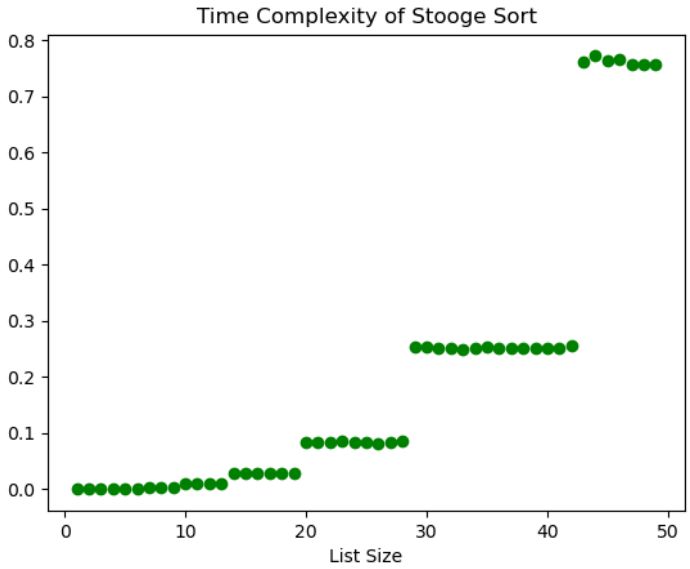
```
def stoogeSort(myList, first, last):                                    O(1)

    if myList at first is greater than myList at last                   O(1)
        initialize a temp variable ot myList at first                   O(1)
        set myList at first to myList at last                           O(1)
        set myList at last to be myList at first                        O(1)

    if if last-first is greater than 1:                                 O(1)

        initialize variable oneThird to be (first-last+1)//3            O(1)

        call stoogeSort with parameters myList, first, last-oneThird    T(3n/2)
        call stoogeSort with parameters myList, first+oneThird, last    T(3n/2)
        call stoogeSort with parameters myList, first, last-oneThird    T(3n/2)

    return myList                                                       O(1)
```

Stooge sort first checks if the first element is larger than the last element. If this is the case they are switched. Then stooge sort is recursively called on the first two thirds of the list. Then on the last two thirds of the list. Finally, it is called again on the first two thirds.

T(n) = 3T(3n/2) + 1
a = 3
b = 3/2
c = 0
log_b(a) = log_3/2(3) = 2.7 > c
So by the master theorem T(n) = O(n^2.7).

For the stooge sort algorithm we do all of our operations in place and no extra memory is allocated so the big O memory requirement is O(1).

The graph looks a little strange but it is reasonable to assume that it has a growth rate of roughly n^2.7 and is consistent with the big O analysis.



```
class Node:                                                    O(1)

    def __init__(self, data):                                  O(1)

        initialize self.left to None                           O(1)
        intitialize self.right to None                         O(1)
        initialize self.data to data                           O(1)

# Insert method to create nodes                                O(1)
    def insert(self, data):                                    O(1)

        if self.data is not None:                              O(1)
            if data is less than self.data:                    O(1)
                if self.left is None:                          O(1)
                    set self.left to a Node of data            O(1)
                else:                                          O(1)
                    insert data to self.left                   T(n-1)
            elif data is greater than self.data:               O(1)
                if self.right is None:                         O(1)
                    set self.right to a Node of data           O(1)
                else:                                          O(1)
                    insert data to self.right                  T(n-1)
        else:                                                  O(1)
            set self.data to data                              O(1)

T(n)    = T(n-1) + 1
        = 1 + 1 + T(n-2)
        = 1 + 1 + 1 + T(n-3)
        = 1 + 1 + 1 + … + 1 + T(n-n) = O(n)


    def sortedList(self, myList):                              O(1)
        if self.left is not None:                              O(1)
            call sortedList on self.left                       T(n/2)
        myList.append(self.data)                               O(1)
        if self.right is not None:                             O(1)
            call sortedList on self.right                      T(n/2)

        return myList                                          O(1)
```

If we assume the tree is split roughly evenly between the right and the left subtrees then each recursive call of sortedList will cut the size of the input in have giving:
T(n) = 2T(n/2) + 1
a = 2, b = 2, c = 0
log_b(a) = log_2(2) = 1>0 so T(n) = O(n^log_2(2)) = O(n)

Alternatively, we could have many elements in one tree and not very many in the other. The extreme of this would be every element in one of the subtrees and 0 in the other so each recursive call reduces the size of the input by 1, but one of the recursive calls will not happen
T(n) = T(n-1) + 1
= 1 + 1 + T(n-2)
= 1 + 1 + 1 + T(n-3)
= 1 + 1 + 1 + … + 1 + T(n-n) = O(n)

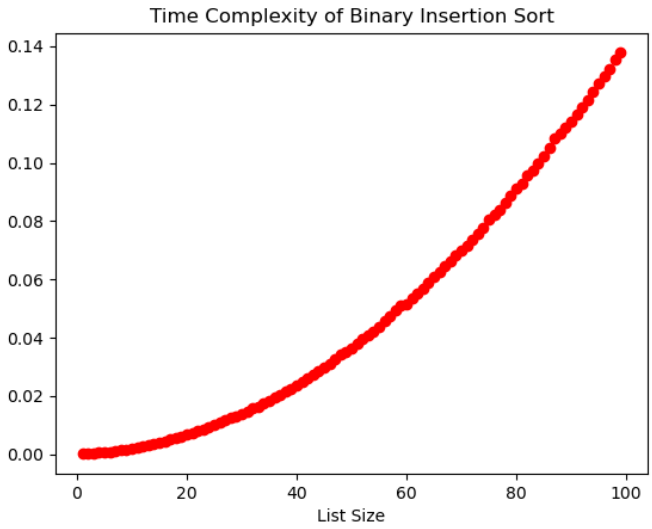
In any case, some elements will be in once subtree and the remaining elements will be in the other one, so there will always be n total between the two. Thus, it makes sense that sortedList will always be O(n)

```
def binaryInsertionSort(myList):                    O(1)
    tree = Node(myList[0])                          O(1)

    for i in range(1, len(myList)):                 O(n)
        tree.insert(myList[i])                      O(n)

    myList = tree.sortedList([])                    O(n)

    return myList                                   O(1)
```

In order to sort the binary insertion sort algorithm creates a binary search tree with the elements from the list. Then it parses through the tree and adds the elements back to the original list in a sorted order.
I did not write the Node class which creates the binary search tree. I found it on tutorialspoint.com at this url
https://www.tutorialspoint.com/python_data_structure/python_binary_search_tree.htm.
The insert function parses through the tree using a recursive call on the appropriate sublist moving right if the node is larger than the element being added and left if it is smaller. Once it finds the correct location it is added as a node to the tree.
I modified the function to print the tree so that it adds the elements back to the list instead. The function parses through the tree, looking for elements to the left of the tree because those will be the smallest, then it adds the furthest left element to the list and moves on to the rest of the tree. Once we parse through the whole tree each element has been added from smallest to largest back to the list completing the sort.

When using the sortedList function in our binaryInsertionSort function we get an overall run time of: O(1) + O(1) + O(n)(O(n)) + O(n) + O(1) = O(n^2)
The first thing the algorithm does is create a binary search tree with n elements. So the memory allocation increases as a linear function of the input size. So the big O memory requirement for this algorithm is O(n)

The big O run time complexity is consistent with the graph below which appears to have a quadratic growth rate relative to the list size.




```
from heapq import heapify, heappush, heappop

def heapSort(myList):                                                          O(1)
    initialize heap to an empty list                                           O(1)
    call heapify on heap                                                       O(n)

    for i from 0 to the length of myList:                                      O(n)
        use heappush to push myList at i onto heap                             O(logn)

    for i from 0 to the length of myList:                                      O(n)
        use heappop to pop from heap and set myList at i to the popped value   O(logn)


    return myList                                                              O(1)
```

The source code for the heapq functions is as follows:
O(n)
def heapify(x):

```
    #Transform list into a heap, in-place, in O(len(x)) time.
    n = len(x)
    # Transform bottom-up.  The largest index there's any point to looking at
    # is the largest with a child index in-range, so must have 2*i + 1 < n,
    # or i < (n-1)/2.  If n is even = 2*j, this is (2*j-1)/2 = j-1/2 so
    # j-1 is the largest, which is n//2 - 1.  If n is odd = 2*j+1, this is
    # (2*j+1-1)/2 = j so j-1 is the largest, and that's again n//2-1.
    for i in reversed(range(n//2)):
        _siftup(x, i)

O(logn)
def heappush(heap, item):
    #Push item onto heap, maintaining the heap invariant.
    heap.append(item)
    _siftdown(heap, 0, len(heap)-1)

def _siftdown(heap, startpos, pos):
    newitem = heap[pos]
    # Follow the path to the root, moving parents down until finding a place
    # newitem fits.
    while pos > startpos:
        parentpos = (pos - 1) >> 1
        parent = heap[parentpos]
        if newitem < parent:
            heap[pos] = parent
            pos = parentpos
            continue
        break
    heap[pos] = newitem"""

O(logn)
def heappop(heap):
    #Pop the smallest item off the heap, maintaining the heap invariant.
    lastelt = heap.pop()    # raises appropriate IndexError if heap is empty
    if heap:
        returnitem = heap[0]
        heap[0] = lastelt
        _siftup(heap, 0)
        return returnitem
    return lastelt

def _siftup(heap, pos):
    endpos = len(heap)
    startpos = pos
```

```
    newitem = heap[pos]
    # Bubble up the smaller child until hitting a leaf.
    childpos = 2*pos + 1    # leftmost child position
    while childpos < endpos:
        # Set childpos to index of smaller child.
        rightpos = childpos + 1
        if rightpos < endpos and not heap[childpos] < heap[rightpos]:
            childpos = rightpos
        # Move the smaller child up.
        heap[pos] = heap[childpos]
        pos = childpos
        childpos = 2*pos + 1
    # The leaf at pos is empty now.  Put newitem there, and bubble it up
    # to its final resting place (by sifting its parents down).
    heap[pos] = newitem
    _siftdown(heap, startpos, pos)
```

I used the heapq library in order to create a min-heap with the elements from the original list. I originally learned of heapq from a GeeksforGeeks post about min heaps in python found here: https://www.geeksforgeeks.org/min-heap-in-python/
I found the documentation for heapq here:
https://docs.python.org/3/library/heapq.html
and the source code here:
https://github.com/python/cpython/blob/3.9/Lib/heapq.py

First we create an empty list and use the heapify function to transform it into a min heap which happens in linear time. A min heap is a binary tree in which every parent node is larger than any of its children. Then we add every element in the list to the heap using heappush. Heappush adds an element to the heap while making sure that every parent remains larger than its children. The heappush function happens in logn time. We then repeatedly use heappop on the heap until every element is removed and added back to the list. Heappop returns and removes the smallest element from the list in logn time.

The overall run time is O(1) + O(1) + O(n) + O(n)(O(logn)) + O(n)(O(logn)) + O(1)

So this heapSort implementation is O(nlogn).
We need to allocate memory for the creation of the heap which will always be the same size as the list. So the big O memory requirement of heapSort is O(nlogn).

A big O run time of O(nlogn) is reasonable to assume for the following graph of run time

```
import random                                                          O(1)
def bogoSort(myList):                                                  O(1)

    initialize keepGoing to true                                       O(1)
    while keepGoing is true:                                           O(n!)
        using random.sample create a random permutation of myList\     O(1)
        and set myList to be this new permutation
        initialize isSorted to True                                    O(1)
        for i from 0 to 1 less than the length of myList               O(n)
            if myList at i is greater than the next element            O(1)
                set isSorted to False                                  O(1)
        if isSorted is True:                                           O(1)
            set keepGoing to False                                     O(1)

    return myList                                                      O(1)
```

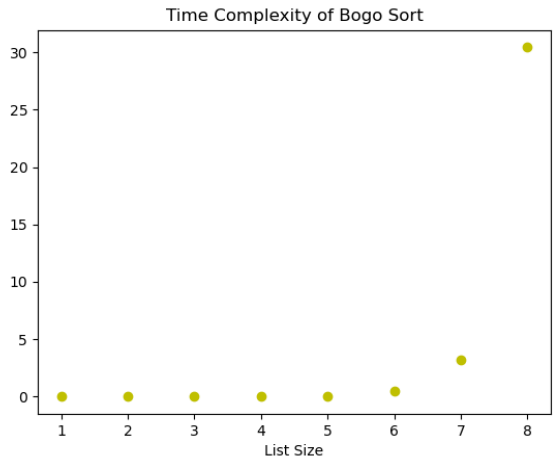
Bogo sort repeatedly creates random permutations of a list until it creates a sorted list. My implementation uses the random.sample function from the python random library to create a random permutation of the list. Then it parses through the randomly generated list checking to see if any element is greater than the one immediately following it. If it finds any unsorted elements the loop restarts and creates a new permutation. If no unsorted elements are found the list must be sorted and the loop terminates and the function returns the list.

The probability of getting a sorted list from a random permutation is 1/n! since there are n! permutations of a list of length n. So, the average case is expected to take n! tries to get a sorted list. So the overal run time comes from the while loop that repeats n! times on average and the for loop that checks if the list is sorted.

So the overall run time is O(n*n!) on average. Since there is no actual upper bound the worst case run time is O(infinity) and since it is theoretically possible to produce a sorted list on the first attempt the best case run time is O(1).

This bogo sort implementation does not create any new data structures so no memory needs to be allocated and the big O memory requirement is O(1)

This algorithm is extremely slow so I could not test it for large values but we can clearly see that the growth rate is extremely high. So it is reasonable to assume that the run times shown here are increasing as a function of n! where n is the size of the list being sorted.



```
def quickSort(myList, start, end):                                  O(1)
    if end is greater than start:                                   O(1)

        initialize pivot to the myList at index start               O(1)

        initialize stop to False                                    O(1)
        initialize i to start+1                                     O(1)
        initialize j to end                                         O(1)
        while stop is False:                                        O(n)(Since this while
loop is moving i and j closer to each other and stops when they cross the entire loop has a O(n)
run time)

            while myList at i is less than pivot and i is less than j:      O(n)
                add 1 to i                                                  O(1)

            while myList at j is greater than pivot and j is greater than i:    O(n)
                subtract 1 from i                                           O(1)

            if i is less than j and myList at i is greater than pivot and\      O(1)
            myList at j is less than pivot:
                initialize temp to myList at i                              O(1)
                set myList at i to myList at j                              O(1)
                set myList at j to temp                                     O(1)

            else:                                                           O(1)
                if myList at j is less than myList at start:                O(1)
                    initialize temp to myList at start                      O(1)
                    set myList at start to myList at j                      O(1)
                    set myList at j to temp                                         O(1)
                set stop to True                                            O(1)

        call quick sort with parameters myList, start, j-1                  T(1)
        call quickSort with parameters myList, i , end                      T(n-1)
```

The quick sort algorithm picks the first element of the list to be a pivot value. It then parses the list from the left looking for an element that is large than the pivot and from the right looking for an element that is smaller than the pivot. If both of these elements are found the algorithm swaps them and continues parsing. If the left and right positions cross each other that means the position where the pivot must go in the list has been found. The algorithm puts the pivot into its correct position which leaves us with a sublist to the left in which every element is less than the pivot and a sublist on the right in which every element is greater than the pivot. We can then call quick sort on the left and right sublists to continue sorting the list.

The worst case run time comes when the list is partitioned into a list of no elements and a list of n-1 elements every time, giving the following run time:

T(n) = T(n-1) + n
= n + T(n-1)
= n + (n + T(n-2))
= n + n + (n + T(n-3))
= n + n + n + … + n + T(n-n) = n^2

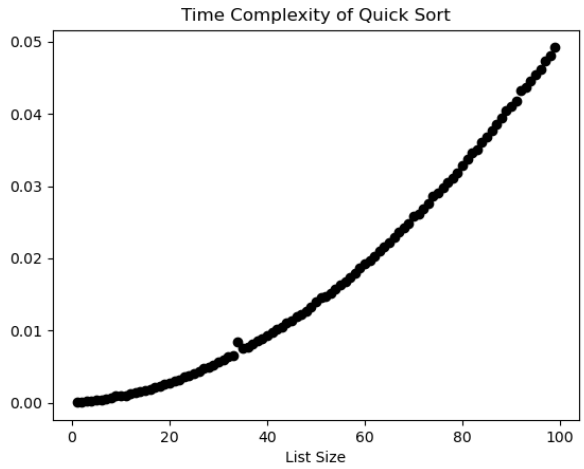
So the worst case run time is O(n^2)

The best case happens when the list is partitioned evenly into two sublists where the run time is
T(n) = 2T(n/2) + n
a = 2
b = 2
c = 1
log_b(a) = log_2(2) = 1 = c so T(n) = O(nlogn)

The average case run time is given by our textbook as O(nlogn) as well.

The quick sort algorithm is done in place without the creation of any extra data structures so it has a big O memory requirement of O(1).

The graph below shows the worst case run time of O(n^2)



In conclusion, in terms of run time merge sort and heap sort seem to be the fastest with big O run times of O(nlogn). Quicksort also has an average run time of O(nlogn) but it has the issue of the worst case run time being O(n^2). If memory is not an issue merge sort or heap sort should be used. If memory is a problem quick sort can be substituted because it is the fasted algorithm that does not require extra memory allocation.