**Project 2 Report – Bret Kagebein, Justin Morgan, Gavin Smith**

In Project 2, we implemented, experimented with and took records of four different sorting algorithms. We created arrays of varying sizes (100, 1000, 10000 and 100000 elements) and in various preset states - ascending order, random order (run 10 times and averages for the purpose of our data), alternating order and descending order - in a set of methods labelled “Case Scenarios”. We implemented the three standard codes – bubble sort, quick sort and merge sort – as well as our choice of the insertion sort. We put each algorithms – and their partner algorithms in a few cases - in a set of methods we labelled “Sort Algorithms”. We imported both of these to be referenced in our main file: our testing suite. Justin did the majority of the scripting and commenting, Gavin ran and executed the tests and troubleshot the code and Bret analyzed the data, calculated O-notations and wrote up the report.

**The Bubble Sort**

The bubble sort is a simple sorting algorithm that is great for smaller groups or for well-sorted groups but becomes drastically less efficient the bigger and less-simply sorted.

The algorithm is shown and marked below:

def bubbleSort(self, array):

arrayLength = len(array) #Evaluate Array

for pass\_num in range(arrayLength):

swapped = False

for i in range(1, arrayLength - pass\_num): #Compare

if array[i - 1] > array[i]:

array[i], array[i - 1] = array[i - 1], array[i] #Swap if first element is larger

swapped = True

if not swapped: #Stop if no swaps occurred

break

return array #Return

Below are the recorded statistics and graphs illustrating how long the algorithm took to sort each size array in each scenario.

First, the best case scenario (Ascending Order):

A graph with a line

Description automatically generated

Array Size 100

Time Taken: 0.00001470 seconds

Array Size 1000

Time Taken: 0.00007290 seconds

Array Size: 10000

Time Taken: 0.00067410 seconds

Array Size: 100000

Time Taken: 0.00413030 seconds

The average case scenario (Random Order):

A graph with a red line

Description automatically generated

Array Size: 100

Time Taken: 0.00347910 seconds

Array Size: 1000

Time Taken: 0.03099585 seconds

Array Size: 10000

Time Taken: 3.67103577 seconds

Array Size: 100000

Time Taken: 441.88322949 seconds

The worst case scenario (Descending Order):

**A graph with a red line

Description automatically generated**

Array Size: 100

Time Taken: 0.00053990 seconds

Array Size: 1000

Time Taken: 0.05067050 seconds

Array Size: 10000

Time Taken: 4.70652870 seconds

Array Size: 100000

Time Taken: 472.54767480 seconds

Across the board, the time rises significantly as the array size increases, and as the order of the elements becomes more and more complex.

Best-Case O-Notation = O(N) – Every element is passed through exactly once.

Average-Case O-Notation = O(N^2) – Nested for-loop

Worst-Case O-Notation = O(N^2) – Nested for-loop (running through every element, every time)

**Merge Sort**

The merge sort is a little more complex sorting algorithm that is consistent for every case.

Merge Sort Algorithm:

def mergeSort(self, array):

if len(array) <= 1: #Evaluate Array

return array

middle = len(array) // 2 # Split list, Sort both halves, and

left = self.mergeSort(array[:middle]) merge them

right = self.mergeSort(array[middle:])

return self.merge(left, right) # Merge the sorted halves

The merge sort algorithm also employs the use of a merge algorithm, shown and marked below:

Merge Function: #Helper function for mergeSort to

def merge(self, left, right): merge the two sorted list

result = []

left\_idx, right\_idx = 0, 0 #Set Indexes

while left\_idx < len(left) and right\_idx < len(right): #Merge the two sorted list

if left[left\_idx] <= right[right\_idx]: #Compare Elements

result.append(left[left\_idx])

left\_idx += 1

else:

result.append(right[right\_idx])

right\_idx += 1

if left\_idx < len(left): #Append remaining left elements

result.extend(left[left\_idx:])

if right\_idx < len(right): #Append remaining right elements

result.extend(right[right\_idx:])

return result #Return (back to Sort Algorithm)

Below are the recorded statistics and graphs illustrating how long the algorithm took to sort each size array in each scenario.

Best case scenario (Ascending Order):

**A graph with a line going up

Description automatically generated**

Array Size: 100

Time Taken: 0.00017090 seconds

Array Size: 1000

Time Taken: 0.00165400 seconds

Array Size: 10000

Time Taken: 0.01198780 seconds

Array Size: 100000

Time Taken: 0.11020630 seconds

The average case (Random Order):

**A graph with a red line

Description automatically generated**

Array Size: 100

Time Taken: 0.00026340 seconds

Array Size: 1000

Time Taken: 0.00300002 seconds

Array Size: 10000

Time Taken: 0.02199912 seconds

Array Size: 100000

Time Taken: 0.20298648 seconds

The worst-case (Alternating Order):

A graph with a red line

Description automatically generated

Array Size: 100

Time Taken: 0.00018330 seconds

Array Size: 1000

Time Taken: 0.00146910 seconds

Array Size: 10000

Time Taken: 0.01162860 seconds

Array Size: 100000

Time Taken: 0.26389000 seconds

The results are close across the board, with consistent, tiny differences in number of elements and even smaller differences between cases.

O-Notation for all cases: O(N log N) – for each element, go through an exponentially decaying number of other elements.

**Quick Sort**

Like the merge sort, the quick sort is another slightly more complex sorting algorithm – mostly due to its recursive nature. In our version, we implement an iterative quicksort because, when testing worst-case at the highest N, we kept hitting the recursion limit. Both the quick sort algorithm and are iterative Quick Sort algorithm are shown below:

Quick Sort Algorithm:

def quickSort(self, array, begin=0, end=None):

if end is None: #Find End point

end = len(array) - 1

return self.iterativeQuickSort(array) #Begin iterative QuickSort process

def iterativeQuickSort(self, array):

stack = [(0, len(array) - 1)] #Initializes a stack with a tuple

while stack: containing the range array

begin, end = stack.pop()

if begin >= end:

continue

pivot\_idx = self.quickSort\_partition(array, begin, end) #Partition array and set pivot

stack.append((begin, pivot\_idx - 1)) # Add subarrays

stack.append((pivot\_idx + 1, end))

Below are the recorded statistics and graphs illustrating how long the algorithm took to sort each size array in each scenario.

The best-case (Random Order):

**A graph with a line drawn on it

Description automatically generated**

Array Size: 100

Time Taken: 0.00020450 seconds

Array Size: 1000

Time Taken: 0.00237470 seconds

Array Size: 10000

Time Taken: 0.01197430 seconds

Array Size: 100000

Time Taken: 0.16403330 seconds

The average-case (Random Order):

**A graph with a red line

Description automatically generated**

Array Size: 100

Time Taken: 0.00099707 seconds

Array Size: 1000

Time Taken: 0.00200081 seconds

Array Size: 10000

Time Taken: 0.07197905 seconds

Array Size: 100000

Time Taken: 0.38985062 seconds

The worst-case (Alternating Order):

**A graph with a red line

Description automatically generated**

Array Size: 100

Time Taken: 0.00078930 seconds

Array Size: 1000

Time Taken: 0.02536750 seconds

Array Size: 10000

Time Taken: 2.67927660 seconds

Array Size: 100000

Time Taken: 251.54848070 seconds

Like merge sort, the times are all fairly close – until the worst-case scenario, where the time jumps exponentially

Best-Case O-Notation = O(N log N) – for each element, go through an exponentially decaying number of other elements.

Average-Case O-Notation = O(N log N) – Same as Best case

Worst-Case O-Notation = O(N^2) – In the worst-case, the pivot need to swap through every single element once for each element.

**Insertion Sort**

The Insertion sort is based on the idea that you take an array, and rebuild a new array inserting elements in ascending order, one-at-a-time from the old array:

Insertion Sort:

def insertionSort(self, array):

for i in range(1, len(array)):

key = array[i] #Clone Element

j = i – 1 #Create search index

while j >= 0 and key < array[j]: #Sort to find smallest

array[j + 1] = array[j] #Swap previous and index

j -= 1

array[j + 1] = key

Below are the recorded statistics and graphs illustrating how long the algorithm took to sort each size array in each scenario.

The best-case (Ascending Order):

**A graph with a line

Description automatically generated**

Array Size: 100

Time Taken: 0.00002130 seconds

Array Size: 1000

Time Taken: 0.00013150 seconds

Array Size: 10000

Time Taken: 0.00067010 seconds

Array Size: 100000

Time Taken: 0.00841840 seconds

The average-case (Random Order):A graph with a red line

Description automatically generated

Array Size: 100

Time Taken: 0.00022820 seconds

Array Size: 1000

Time Taken: 0.01497555 seconds

Array Size: 10000

Time Taken: 1.62276173 seconds

Array Size: 100000

Time Taken: 354.48626161 seconds

The worst-case (Descending Order):**A graph with a red line

Description automatically generated**

Array Size: 100

Time Taken: 0.00033660 seconds

Array Size: 1000

Time Taken: 0.02892970 seconds

Array Size: 10000

Time Taken: 2.84078630 seconds

Array Size: 100000

Time Taken: 294.27693260 seconds

This algorithm runs very similarly to the bubble sort, for an obvious reason: they operate by, in their own slightly different ways, go through and check each element against every other element.

Best-Case O-Notation = O(N) – Every element is passed through exactly once.

Average-Case O-Notation = O(N^2) – Nested for-loop

Worst-Case O-Notation = O(N^2) – Nested for-loop (running through every element, every time)

**Summary**

In their own ways, each code has their own strengths. The bubble sort and the insert sort, while inefficient for very random and very large sets of numbers, is simple to code and are actually the faster the closer to sorted the array is. Maybe in a situation where you are only adding one or a few elements at the end of an already sorted array, these may be best. On the other hand, the merge sort and the quick sort are more consistently better with large unsorted arrays – though in the worst case, quick sort is less efficient than merge.