LAB 2

Spencer Neveux

```
import random
import struct
import time
from statistics import mean

# Method to collect name of file to open
def collect_input():
	user_input = input('Enter name of file to open: ')
	return user_input

# Method to read binary file
def read_bin(file_name):
	original = []
	with open(file_name, 'rb') as file:
		data = file.read()
		size = struct.unpack('i', data[0:4])[0]
		for i in range(4, size*4+4, 4):
			element = struct.unpack('i', data[i:i+4])
			original.append(element[0])
		return original

# Methods to create binary files
def create_random_bin():
	NUM_INTEGERS = 10000
	with open('random.bin', 'wb') as file:

		file.write(struct.pack('i', NUM_INTEGERS))

		for i in range(NUM_INTEGERS+1):
			file.write(struct.pack('i', random.randint(0, 1000000000)))

def create_sorted_bin():
	NUM_INTEGERS = 10000
	with open('sorted.bin', 'wb') as file:

		file.write(struct.pack('i', NUM_INTEGERS))

		for i in range(1, NUM_INTEGERS+1):
			file.write(struct.pack('i', i))
```

```
def create_reverse_bin():
	NUM_INTEGERS = 10000
	with open('reverse.bin', 'wb') as file:

		file.write(struct.pack('i', NUM_INTEGERS))

		for i in reversed(range(1, NUM_INTEGERS+1)):
			file.write(struct.pack('i', i))


# Method to check if array is in non-decreasing order
def is_ordered(array):
	for i in range(1, len(array)):
		if array[i-1] > array[i]:
			return False
	return True

# Method to find the median of three
def median3(array, left, right):
   mid = (left+right-1)//2
   a = array[left]
   b = array[mid]
   c = array[right-1]
   if a <= b <= c:
      return mid
   if c <= b <= a:
      return mid
   if a <= c <= b:
      return right-1
   if b <= c <= a:
      return right-1
   return left

# Method to partition list
def partition(array, left, right, pi):
	swap(array, pi, right)
	pivot_value = array[right]
	store = left
	for i in range(left, right):
		if array[i] <= pivot_value:
			swap(array, store, i)
			store = store+1
	swap(array, store, right)
	return store
```

```
# Method to swap elements in list
def swap(array, pos1, pos2):
	array[pos1], array[pos2] = array[pos2], array[pos1]
	return array

# Insertion Sort method
def insertion_sort(array, left, right):
	for i in range(left+1, right+1):
		j2 = i-1
		temp = array[i]
		for j in range(i-1, left-2, -1):

			j2 = j
			if j == left-1:
				break

			if array[j] > temp:
				array[j+1] = array[j]
			else:
				break
		array[j2+1] = temp
	return array

# Quick Sort method
def quickSort(array, left, right):
	if (right-left) < 10:
		insertion_sort(array, left, right)
	if left < right:
		pi = median3(array, left, right)
		pivot_index = partition(array, left, right, pi)
		quickSort(array, left, pivot_index-1)
		quickSort(array, pivot_index+1, right)

# ------------
# Buffer VM
# Methods to prepare virtual machine
# ------------
def buffer_insertion(file_name):
	array_sorted = read_bin(file_name)
	for i in range(5):
		insertion_sort(array_sorted, 0, len(array_sorted) -1)

def buffer_quick():
	array_sorted = read_bin(file_name)
	for i in range(5):
		quick_sort(array_sorted, 0, len(array_sorted)-1)
```

```
# ------------
# Test
# ------------
def main():
	# # Create binary files
	create_sorted_bin()
	create_reverse_bin()
	create_random_bin()

	# Collect file_name to be read
	file_name = collect_input()
	original = read_bin(file_name)

	# Make copies of original array
	insertion_array = original[:]
	quick_array = original[:]

	# # Time insertion sort
	time_i = []
	for i in range(10):
		print(f'Array Ordered: {is_ordered(insertion_array)}')
		time_1 = time.clock()
		insertion_array = insertion_sort(insertion_array, 0, len(insertion_array)-1)
		elapsed_time = time.clock() - time_1
		time_i.append(elapsed_time)
		print(f'Array Ordered: {is_ordered(insertion_array)}, Time Taken Insertion Sort:
{format(elapsed_time, ".3e")}')
		insertion_array = original[:]
	print(f'Sorted time array {time_i}, Avg = {mean(time_i)}')

	# # Time quick sort
	time_q = []
	for i in range(10):
		print(f'Array Ordered: {is_ordered(quick_array)}')
		time_1 = time.clock()
		quickSort(quick_array, 0, len(quick_array) -1)
		elapsed_time = time.clock() - time_1
		time_q.append(elapsed_time)
		print(f'Array Ordered: {is_ordered(quick_array)}, Time Taken Quick Sort:
{format(elapsed_time, ".2e")}')
		quick_array = original[:]
	print(f'Sorted time array {time_q}, Avg = {mean(time_q)}')

if __name__ == '__main__':
	main()
```

Analysis:

We know that insertion sort has a big-oh value of O(n^2), while quicksort has a big-oh value of O(nlogn). We can see that insertion sort will be fine for sizes of n that are sufficiently small, however, the performance will decrease dramatically as n increases. We also know that insertion sort is an adaptive algorithm, meaning that it's performance will increase for lists that are already sorted. This is not the case for quicksort, in fact, quicksort will experience its worst case run time on a data set that is already sorted.

1) Knowing that insertion sort is an adaptive algorithm, we can guess that it will run the fastest on the sorted.bin file. This is due to the inherent nature of insertion sort. The algorithm will run through the array and compare a[i-1] <= a[i]. If the elements are not in non-decreasing order, the values are swapped. Since we won't need to swap any of the elements in a sorted array, the algorithm will run in its best time for a sorted array. **Prediction** - **Sorted.bin**

2) Following the same logic as above, it is easy to guess that the algorithm will experience its worst performance on an array that is in reverse order. This is because it's going to need to swap every element in the array multiple times. The element at a[0] will need to be moved all the way to a[n-1]. Then the element a[1] will need to move all the way to a[n-2], etc. This will surely take longer than sorting an array of elements in random order, as it's likely they won't all need to be swapped. **Prediction** - **Reverse.bin**

3) We know that quicksort performs by identifying a pivot value and partitioning the array accordingly around that pivot. Now if the array happened to be in sorted order whether that's increasing or decreasing, the algorithm will degenerate into a run time complexity of O(n^2). This is because every time a pivot is selected, the pivot value is moved to the right most position in the array. Then we iterate through the array and compare each value to that pivot. If the value being examined is greater than the pivot value, then we shift its position to the right in the array. However, if the value is less than or equal to the pivot, we simply leave it as it is. Now if the array is already sorted, we are doing unnecessary work. So the fastest case for quicksort would on average be an array that is in random order. **Prediction - Random.bin**

4) We know that insertion sort actually performs better on partially sorted arrays, and that quicksort experiences its worst performance on sorted arrays.
Prediction - Insertion Sort

5)

Insertion Sort - Time

											Avg
Sorted (ms)	**7.83**	**6.79**	**7.11**	**7.66**	**6.73**	**6.85**	**7.21**	**7.22**	**6.74**	**7.21**	**7.16**
Random (ms)	**5218**	**5936**	**5318**	**5832**	**5385**	**5803**	**5916**	**5900**	**5944**	**5634**	**5724**
Reverse- (ms)	**1038**	**1100**	**1130**	**1062**	**1082**	**1082**	**1093**	**1086**	**1082**	**1130**	**1088**

Quick Sort - Time

											Avg
Sorted- (ms)	**52.2**	**51.3**	**51.2**	**50.2**	**51.7**	**55.0**	**53.4**	**56.0**	**53.9**	**55.6**	**53.0**
Random - (ms)	**55.9**	**55.4**	**72.7**	**64.1**	**5.9**	**58.9**	**58.2**	**59.6**	**68.5**	**56.3**	**60.1**
Reverse- (ms)	**79.1**	**79.8**	**80.6**	**79.8**	**79.6**	**78.8**	**83.1**	**81.6**	**80.4**	**81.1**	**80.3**

6) As we can see, the first prediction was spot on. Insertion sort ran exponentially faster on a sorted array of numbers than it did on randomly sorted, or reverse sorted. We can also clearly see that insertion sort performed the worst on an array of numbers that were in reverse order, just as predicted. If we examine the run times of quick sort, we can see that the prediction was not in fact correct. Even though it was assumed quick sort would run fastest on a randomly sorted array, it turns out the performance was not much better than on a sorted array. In fact, quicksort ran faster on a sorted array. It would seem this diversion from

expected performance might be due to the median of three function. By choosing the median of the first, middle, and last element, we can in fact ensure that fully sorted data remains optimal. Which would appear to be the case here. Finally, if we look at the run times for insertion sort versus quick sort, we can see that insertion sort did much better on a sorted array of values. Coinciding with perfectly with the prediction.