Ocean Lu
CS 3310
Professor Johannsen
10/03/2019

Project #1: Typed Report

NOTE Please view my google colab file, populated with all my implementations and tests: https://colab.research.google.com/drive/1xdrHHqchwUpGQlh8dRpBg5tgRgCMMYPF
(same as attached source code)

1. Explain how you verified the correctness of your implementation of the five algorithms.
   I implemented the following algorithms in Python 3: To test the implementation of my five algorithms, I tested using empirical analysis, which utilizes experimentation and observation of results.
   a. Insertion Sort:

```
index location (index low, index high)
{
  index mid;

  if (low > high)
    return 0;
  else {
    mid = ⌊(low + high)/2⌋;
    if (x == S[mid])
      return mid
    else if (x < S[mid])
      return location(low, mid − 1);
    else
      return location(mid + 1, high);
  }
}
```

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
```

   b. Mergesort:

```
void mergesort (int n, keytype S[])
{
  if (n>1) {
     const int h = ⌊ n/2⌋, m = n − h;
     keytype U[1..h], V[1..m];
     copy S[1] through S[h] to U[1] through U[h];
     copy S[h+1] through S[n] to V[1] through V[m];
     mergesort(h, U);
     mergesort(m, V);
     merge(h, m, U, V, S);
  }
}
```

```
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1
        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1

        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
```

c. Quicksort1 (Regular Quicksort):

```
void quicksort (index low, index high)
{
  index pivotpoint;

  if (high > low){
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint − 1);
     quicksort(pivotpoint + 1, high);
  }
}
```

```python
# This function takes last element as pivot
def partition(arr,low,high):
    i = (low-1)
    pivot = arr[high]
    for j in range(low , high):
        if   arr[j] <= pivot:
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]
    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )
def quickSort1(arr,low,high):
    if low < high:
        pi = partition(arr,low,high)
        quickSort1(arr, low, pi-1)
        quickSort1(arr, pi+1, high)
```

d. Quicksort2 (Quicksort / Insertion Sort Combo):

```python
def partition(arr,l,h):
    i = ( l - 1 )
    x = arr[h]
    for j in range(l , h):
        if   arr[j] <= x:
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]
    arr[i+1],arr[h] = arr[h],arr[i+1]
    return (i+1)
def quickSort2(arr,l,h):
    size = h - l + 1
    stack = [0] * (size)
    top = -1
    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h
    while top >= 0:
        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1
        p = partition( arr, l, h )
        if p-1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p - 1
        if p+1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h
```

e. Quicksort3 (Randomized Quicksort):

```python
import random
def quickSort3(arr, start, stop):
    if(start < stop):
        pivotindex = partitionrand(arr, start, stop)
        quickSort3(arr , start , pivotindex)
        quickSort3(arr, pivotindex + 1, stop)
def partitionrand(arr , start, stop):
    randpivot = random.randrange(start, stop)
    arr[start], arr[randpivot] = arr[randpivot], arr[start]
    return partition(arr, start, stop)

def partition(arr,start,stop):
    pivot = start
    i = start - 1
    j = stop + 1
    while True:
        while True:
            i = i + 1
            if arr[i] >= arr[pivot]:
                break
        while True:
            j = j - 1
            if arr[j] <= arr[pivot]:
                break
        if i >= j:
            return j
        arr[i] , arr[j] = arr[j] , arr[i]
```

2. Write a detailed report together with tables describing the data sets, test strategies, and performance results.

For the data sets, I randomly generated an array for each of the algorithms. Meaning that each algorithm had their own unique set of data that ranges from 0 to 1, unique to the array size. For example, if the array size was 2^5 (32), then I would generate an array with 32 random elements and have a specific sorting algorithm (let's say Insertion Sort) sort it. After Insertion Sort has finished sorting and I have correctly timed it, I move onto generating a new array sized 32, with random elements (different from Insertion Sort's array) and have another specific sorting algorithm sort it (the next one would be Merge Sort). The array size stays the same until all five algorithms have completed sorted and will be multiplied by 2 (respecting the exponential increments with basis of two).

```
def makeArr(n):
  array = np.random.rand(n)
  return array
```

For test strategies, I implemented a while loop that would iterate 16 times, with the array sizes 2^1, 2^2, 2^3, …, 2^{14}, 2^{15}, 2^{16}. In each iteration, I would make sure that I initialize the array with the specific array size, start a timer before I sort the array, sort the array, and then end the timer. By doing so, I can subtract the time it ended at to the time it started at, concluding with the performance results I needed to analyze. This type of behavior is implemented on all the sorted algorithms and has their own individual timer. Even though this test practice may be relatively complicated (ex: creating the randomized array list), I can make sure that the time is independent of that through my sequential declaration.

```
arrSize = 2
counter = 1
while (arrSize < 65537):
  print ("array size 2^", counter, "or", arrSize)

  # insertionSort
  arr = makeArr(arrSize)
  t0 = time.time()
  insertionSort(arr)
  t1 = time.time()
  total = t1-t0
  print ("insertionSort:", total)

  # mergeSort
  arr = makeArr(arrSize)
  t0 = time.time()
  mergeSort(arr)
  t1 = time.time()
  total = t1-t0
  print ("mergeSort:", total)

  # quickSort1 (regular)
  arr = makeArr(arrSize)
  t0 = time.time()
  quickSort1(arr,0,len(arr)-1)
  t1 = time.time()
  total = t1-t0
  print ("quickSort1 (regular):", total)

  # quickSort2 (insertion)
  arr = makeArr(arrSize)
  t0 = time.time()
  quickSort2(arr,0, (len(arr)-1))
  t1 = time.time()
  total = t1-t0
  print ("quickSort2 (insertion):", total)

  # quickSort3 (randomized)
  arr = makeArr(arrSize)
  t0 = time.time()
  quickSort3(arr,0, (len(arr)-1))
  t1 = time.time()
  total = t1-t0
  print ("quickSort3 (randomized):", total)

  arrSize = arrSize * 2
  counter = counter + 1
  print()
```

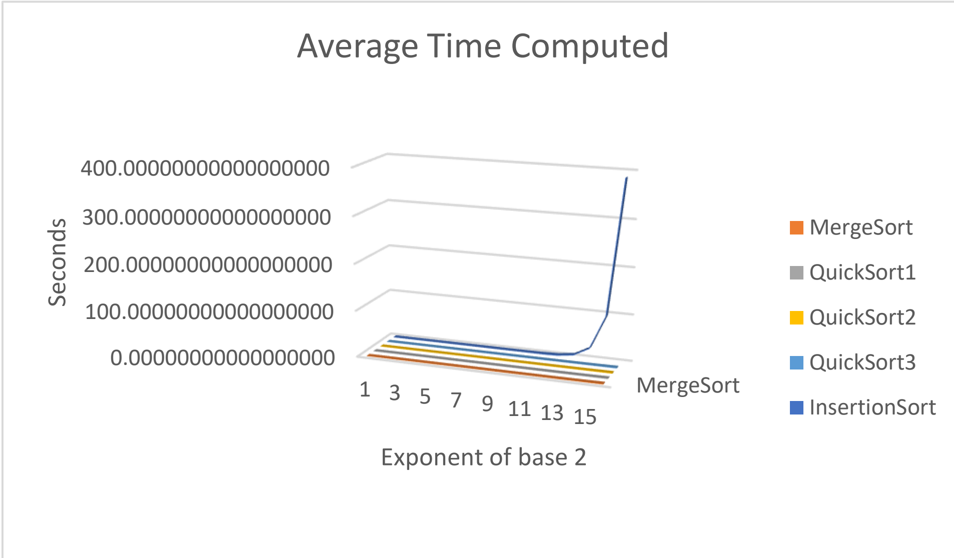
For performance results, things are printed even before the task finishes going through the entire while loop. Our output has several things, but not the average as I do tests through iterating with the array size and calculated the average by hand. I chose this method because it intuitively checks every sorting algorithm with a random case scenario. I also felt the need to not overcomplicate things by adding another array to store the average and calculating it later, although it could be easily implemented. In any case, the first line is what specific array size we are testing, and then the sorting algorithm and its respective time to how long it had gone through the process.

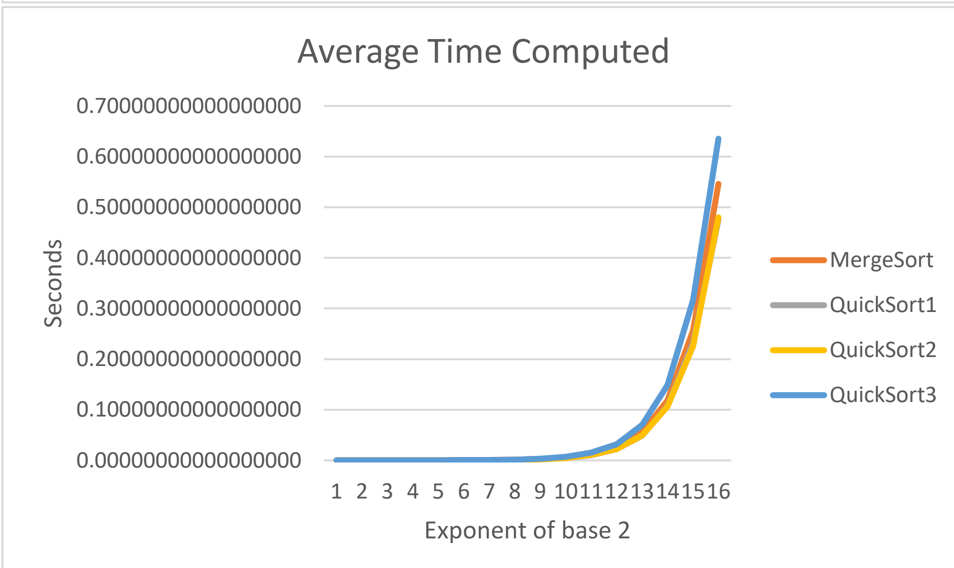
```
array size 2^ 1 or 2
insertionSort: 1.4781951904296875e-05
mergeSort: 8.106231689453125e-06
quickSort1 (regular): 5.245208740234375e-06
quickSort2 (insertion): 6.198883056640625e-06
quickSort3 (randomized): 2.8133392333984375e-05

array size 2^ 2 or 4
insertionSort: 1.1205673217773438e-05
mergeSort: 2.0265579223632812e-05
quickSort1 (regular): 1.3828277587890625e-05
quickSort2 (insertion): 1.2159347534179688e-05
quickSort3 (randomized): 3.4809112548828125e-05

array size 2^ 3 or 8
insertionSort: 1.9073486328125e-05
mergeSort: 3.719329833984375e-05
quickSort1 (regular): 2.6464462280273438e-05
quickSort2 (insertion): 3.600120544433594e-05
quickSort3 (randomized): 5.650520324707031e-05

array size 2^ 4 or 16
insertionSort: 0.0001049041748046875
mergeSort: 8.082389831542969e-05
quickSort1 (regular): 5.555152893066406e-05
quickSort2 (insertion): 7.796287536621094e-05
quickSort3 (randomized): 0.00011396408081054688

array size 2^ 5 or 32
insertionSort: 0.00013685226440429688
mergeSort: 0.0001347064971923828
quickSort1 (regular): 0.00010347366333007812
quickSort2 (insertion): 0.00011134147644042969
quickSort3 (randomized): 0.0002493858337402344

array size 2^ 6 or 64
insertionSort: 0.0003845691680908203
mergeSort: 0.0002422332763671875
quickSort1 (regular): 0.00023984909057617188
quickSort2 (insertion): 0.0001766681671142578
quickSort3 (randomized): 0.00033092498779296875

array size 2^ 7 or 128
insertionSort: 0.0014698505401611328
mergeSort: 0.0005433559417724609
quickSort1 (regular): 0.0004057884216308594
quickSort2 (insertion): 0.00041413307189941406
quickSort3 (randomized): 0.0007233619689941406

array size 2^ 8 or 256
insertionSort: 0.005746364593505859
mergeSort: 0.001157999038696289
quickSort1 (regular): 0.0008933544158935547
quickSort2 (insertion): 0.0009222030639648438
quickSort3 (randomized): 0.0014569759368896484
```

```
array size 2^ 9 or 512
insertionSort: 0.02298450469970703
mergeSort: 0.00264739990234375
quickSort1 (regular): 0.002172708511352539
quickSort2 (insertion): 0.0023488998413085938
quickSort3 (randomized): 0.003305673599243164

array size 2^ 10 or 1024
insertionSort: 0.09591913223266602
mergeSort: 0.005637407302856445
quickSort1 (regular): 0.00493621826171875
quickSort2 (insertion): 0.004535436630249023
quickSort3 (randomized): 0.00799417495727539

array size 2^ 11 or 2048
insertionSort: 0.3748183250427246
mergeSort: 0.011820077896118164
quickSort1 (regular): 0.010532379150390625
quickSort2 (insertion): 0.010340452194213867
quickSort3 (randomized): 0.014928579330444336

array size 2^ 12 or 4096
insertionSort: 1.5076591968536377
mergeSort: 0.025551557540893555
quickSort1 (regular): 0.0238800048828125
quickSort2 (insertion): 0.022886037826538086
quickSort3 (randomized): 0.033414602279663086

array size 2^ 13 or 8192
insertionSort: 5.980031728744507
mergeSort: 0.05582880973815918
quickSort1 (regular): 0.04873156547546387
quickSort2 (insertion): 0.05108332633972168
quickSort3 (randomized): 0.07397675514221191

array size 2^ 14 or 16384
insertionSort: 24.51366686820984
mergeSort: 0.11819720268249512
quickSort1 (regular): 0.11296939849853516
quickSort2 (insertion): 0.11111044883728027
quickSort3 (randomized): 0.14619064331054688

array size 2^ 15 or 32768
insertionSort: 99.07651042938232
mergeSort: 0.2582390308380127
quickSort1 (regular): 0.2293834686279297
quickSort2 (insertion): 0.2416706085205078
quickSort3 (randomized): 0.3155252933502197

array size 2^ 16 or 65536
insertionSort: 396.4576749801636
mergeSort: 0.5447902679443359
quickSort1 (regular): 0.48235344886779785
quickSort2 (insertion): 0.5120372772216797
quickSort3 (randomized): 0.6511597633361816
```

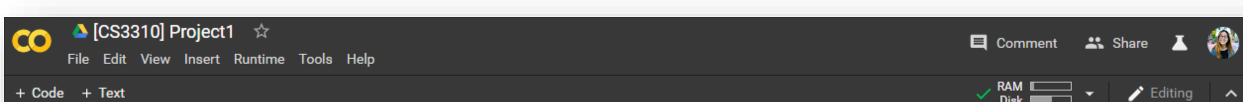
Analysis on output of execution:
In the order of most effective to least effective, it goes: QuickSort1 (regular), QuickSort2 (insertion combo), MergeSort, QuickSort3, and InsertionSort. It looks like QuickSort1 tied with QuickSort2, and almost makes QuickSort1 invisible because QuickSort2 is a layer over QuickSort1. To see the details on calculating the average of 5 tests and making

specific graphs to summarize the data, please view the Output of Execution file!





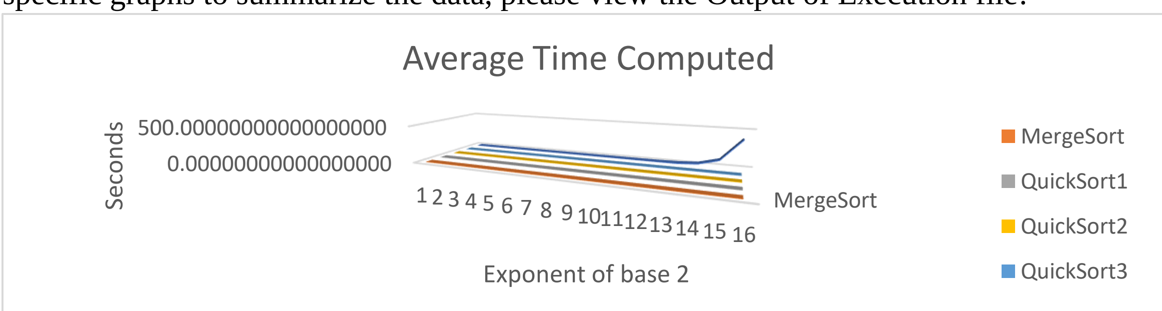
3. What computer did you use for testing?
Colaboratory is a Google research project created to help disseminate machine learning education and research. It's a Jupyter notebook environment that requires no setup to use and runs entirely in the cloud. It provides a single 12GB NVIDIA Tesla K80 GPU that can be used up to 12 hours continuously. Recently, Colab also started offering free TPU. I use a ASUS Laptop 13.3" FHD Display, Intel 8th gen Core i5-8250U, 8GB RAM, 256GB M.2 SSD, on Windows 10.

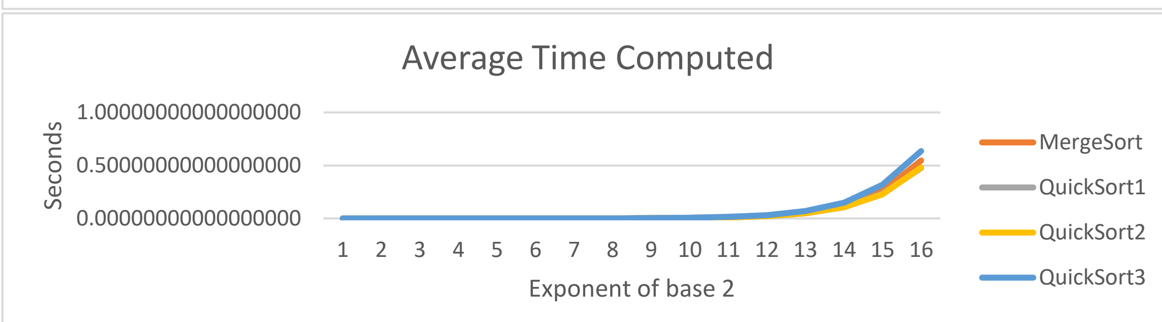


4. Did you try both random inputs and sorted arrays?
Yes, I did indeed try both random inputs and sorted arrays. I used sorted arrays to test and had it as driver code right after I defined the function (edited freely) and used the random inputs for the test strategy in my performance evaluation. For example, for Insertion Sort:

```
[ ]  # a. Insertion Sort

     def insertionSort(arr):
         for i in range(1, len(arr)):
             key = arr[i]
             j = i-1
             while j >=0 and key < arr[j] :
                     arr[j+1] = arr[j]
                     j -= 1
             arr[j+1] = key

     # Driver code
     arr = [12, 11, 13, 5, 6]
     insertionSort(arr)
     print (arr)
```

5. Does Quicksort3 significantly improve the performance of Quicksort1 on sorted arrays?
On a case with sorted arrays, QuickSort3 does significantly improve the performance of Quicksort1.
6. What does your program show about the average execution time of the five algorithms for different sizes of input?
In the order of most effective to least effective, it goes: QuickSort1 (regular), QuickSort2 (insertion combo), MergeSort, QuickSort3, and InsertionSort. It looks like QuickSort1 tied with QuickSort2, and almost makes QuickSort1 invisible because QuickSort2 is a layer over QuickSort1. To see the details on calculating the average of 5 tests and making specific graphs to summarize the data, please view the Output of Execution file!





7. Is Quicksort2/Quicksort3 always faster than Quicksort1?
   Quicksort2 (insertion) compared to Quicksort1 (regular) will only be faster with small lists, as it is very practical. It will not necessarily be always faster. For Quicksort3 (randomized) compared to Quicksort1 (regular), it also depends on the chance of the random its runtime is expected to be O(nlogn), so it won't necessarily always be faster than QuickSort1. Depending on the scenario it won't always mean that QuickSort2/Quicksort3 will always be faster than QuickSort1.
8. What is the relation between the average computing times of Quicksort2 and Quicksort3?
   The relation between the two seem relatively similar, with QuickSort2 becoming more efficient as the data becomes bigger. QuickSort3 grew bigger as the data became bigger, emphasizing how randomizing may not be the most effective choice.
   The average computing time of Quicksort2 and Quicksort3 are:

Average															
2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0.00001626014709473	0.00001678466796875	0.00001730918884277	0.00004444122314453	0.00015029907226563	0.00052747726440430	0.00176029205322265	0.00549569129943847	0.02243709564208980	0.09433040618896480	0.38528704643249500	1.51877508163452000	6.02184967994689000	24.0154840469360000	96.8358266830444000	386.58997077941800000
0.00013217926025391	0.00002670288085937	0.00003466606140137	0.00006799697875977	0.00014925003051758	0.00048375129699707	0.00064978599548340	0.00116453170776367	0.00257081985473632	0.00611429214477539	0.01173539161682120	0.02550120353698730	0.05521440505981440	0.11803426742553700	0.25685071945190400	0.54598851203918400
0.00001420974731445	0.00001382827758789	0.00002536773681641	0.00004816055297852	0.00010671615600586	0.00025968551635742	0.00058670043945313	0.00090742111206055	0.00214905738830566	0.00484924316406250	0.01035051345825190	0.02231464385986330	0.04878835678100580	0.10743455886840800	0.23240451812744100	0.47504696846008200
0.00000953674316406	0.00001921653747559	0.00002541542053223	0.00006170272827148	0.00012183189392090	0.00027751922607422	0.00055646896362305	0.00091934204101562	0.00220193862915038	0.00473442077636718	0.01051692962646480	0.02211842536926260	0.04920511245727530	0.10631842613220200	0.22587056159973100	0.47987384796142500
0.00002551078796387	0.00002784729003906	0.00005698204040527	0.00010209083557129	0.00023307800292969	0.00060048103332519	0.00078730583190918	0.00149397850036621	0.00332226753234863	0.00723958015441894	0.01511383056640620	0.03196272850036620	0.06995186805725090	0.14852008819580000	0.31677141189575100	0.63528370857238700

9. Do your experimental results match with the theoretical analysis of the algorithms?
   My experimental results do match with the theoretical analysis of the algorithms. QuickSorts are faster than MergeSorts, which are faster than InsertionSorts!
10. If not, what are the possible reasons?
    A reason my data would not match would probably be due to my coding, number of tests, and maybe application.
11. Also, find out for what value of n does Quicksort2/Quicksort3 become faster than the other three methods.
    QuickSort2 would be faster if there are small datasets. QuickSort3 would be faster if the chosen random switches perfectly in the middle.