Project 2

CS 3130 – Algorithms

Connor Baniak

Array Setup:

```
list_hundred_sorted = list(range(1, 100))
list_hundred_random = random.sample(range(1, 100), 99)
list_hundred_semiSort = list(range(1,100))

# randomize every 10th pos of array
for i in list_hundred_semiSort:
    if i % 10 != 0 and i != 0:
        continue
    list_hundred_semiSort[i] = randint(1,100)


list_thousand_sorted = list(range(1, 1000))
list_thousand_random = random.sample(range(1, 1000), 999)
list_thousand_semiSort = list(range(1,1000))

# randomize every 10th pos of array
for i in list_thousand_semiSort:
    if i % 10 != 0 and i != 0:
        continue
    list_thousand_semiSort[i] = randint(1,1000)


list_tenthous_sorted = list(range(1, 10000))
list_tenthous_random = random.sample(range(1, 1000), 9999)
list_tenthous_semiSort = list(range(1,10000))

# randomize every 10th pos of array
for i in list_tenthous_semiSort:
    if i % 10 != 0 and i != 0:
        continue
    list_tenthous_semiSort[i] = randint(1,10000)
```

Timer Setup:

```
start = timer()
# call to function
end = timer()

print ("    time:",(end - start))
```

Selection Sort:

```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
```

Size 100 Sorted: time: 0.00030948200037528295

Size 100 Random: time: 0.0003088490002483013

Size 100 Semi-sorted: time: 0.00030423600037465803

Size 1,000 Sorted: time: 0.030191954999281734

Size 1,000 Random: time: 0.029274690999955055

Size 1,000 Semi-sorted: time: 0.02923272000043653

Size 10,000 Sorted: time: 3.253736996999578

Size 10,000 Random: time: 3.1393919709998954

Size 10,000 Semi-sorted: time: 3.073822876000122

Insertion Sort:

```
def insertionSort(alist):
    for index in range(1,len(alist)):

      currentvalue = alist[index]
      position = index

      while position>0 and alist[position-1]>currentvalue:
          alist[position]=alist[position-1]
          position = position-1

      alist[position]=currentvalue
```

Size 100 Sorted:
```
time:
1.3177999790059403e-05
```

Size 100 Random:
```
time:
0.0003178959996148478
```

Size 100 Semi-sorted:
```
time:
4.4829000216850545e-05
```

Size 1,000 Sorted:
```
time:
0.000136313999973936
```

Size 1,000 Random:
```
time:
0.03952427100011846
```

Size 1,000 Semi-sorted:
```
time:
0.005065463999926578
```

Size 10,000 Sorted:
```
time:
0.0012103380004191422
```

Size 10,000 Random:
```
time:
4.062289090999911
```

Size 10,000 Semi-sorted:
```
time:
0.5311772759996529
```

Bubble Sort w/ Swaps:

```python
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already
        #  in place
        for j in range(0, n - i - 1):

            # traverse the array from 0 to
            # n-i-1. Swap if the element
            # found is greater than the
            # next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # IF no two elements were swapped
        # by inner loop, then break
        if swapped == False:
            break
```

Size 100 Sorted:	time: 1.0960000054183183e-05
Size 100 Random:	time: 0.0005250460000070234
Size 100 Semi-sorted:	time: 0.0002915720000373767
Size 1,000 Sorted:	time: 7.63509999615053e-05
Size 1,000 Random:	time: 0.06983441400006996
Size 1,000 Semi-sorted:	time: 0.04271494200020243
Size 10,000 Sorted:	time: 0.0008221870000397757
Size 10,000 Random:	time: 7.32554325000001
Size 10,000 Semi-sorted:	time: 4.198096970000051

Bubble Sort w/o Swaps:

```python
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Size 100 Sorted:	time: 0.00041741700033526286
Size 100 Random:	time: 0.000583070000175212
Size 100 Semi-sorted:	time: 0.00038452899980256916
Size 1,000 Sorted:	time: 0.03547649200027081
Size 1,000 Random:	time: 0.0618706039999779
Size 1,000 Semi-sorted:	time: 0.03768943999966723
Size 10,000 Sorted:	time: 3.6287954330000503
Size 10,000 Random:	time: 6.46160159100009
Size 10,000 Semi-sorted:	time: 4.0814122850001695

Quicksort:

```
def partition(lst, start, end, pivot):
    lst[pivot], lst[end] = lst[end], lst[pivot]
    store_index = start
    for i in range(start, end):
        if lst[i] < lst[end]:
            lst[i], lst[store_index] = lst[store_index], lst[i]
            store_index += 1
    lst[store_index], lst[end] = lst[end], lst[store_index]
    return store_index


def quick_sort(lst, start, end):
    if start >= end:
        return lst
    pivot = randrange(start, end + 1)
    new_pivot = partition(lst, start, end, pivot)
    quick_sort(lst, start, new_pivot - 1)
    quick_sort(lst, new_pivot + 1, end)


def sort(lst):
    quick_sort(lst, 0, len(lst) - 1)
    return lst
```

Size 100 Sorted:

```
time:
0.00025503499989554257
```

Size 100 Random:

```
time:
0.00018410500001664332
```

Size 100 Semi-sorted:

```
time:
0.00015130099995985802
```

Size 1,000 Sorted:

```
time:
0.0022739560000104575
```

Size 1,000 Random:

```
time:
0.0021761340000239215
```

Size 1,000 Semi-sorted:

```
time:
0.0019907390000071246
```

Size 10,000 Sorted:

```
time:
0.027657939999926384
```

Size 10,000 Random:

```
time:
0.02514402400004201
```

Size 10,000 Semi-sorted:

```
time:
0.02614245100005519
```

Mergesort:

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)
```

Size 100 Sorted: time: 0.00016524399961781455

Size 100 Random: time: 0.0001820759998736321

Size 100 Semi-sorted: time: 0.0001661569995121681

Size 1,000 Sorted: time: 0.002291533000061463

Size 1,000 Random: time: 0.0030310319998534396

Size 1,000 Semi-sorted: time: 0.0025943430000552326

Size 10,000 Sorted: time: 0.029053878999548033

Size 10,000 Random: time: 0.037417423999613675

Size 10,000 Semi-sorted: time: 0.036283915000240086

Analysis:

A hypothesis can be made for each of the nine array sizes scenarios when applied to our six sorting algorithms. The arrays range from size 100, 1,000, and 10,000 and are arranged in sorted, random, and semi-sorted fashion. Run times will then be generated and we will analyze the outcomes.

The first scenario to hypothesize is the best case scenario. Insertion Sort and Bubble Sort with swaps are predicted to be the fastest, as their estimated run time Ω(N). The next fastest group of algorithms are Quicksort and Mergesort with an estimated runtime of Ω(N LOG N). Lastly, Selection Sort and Bubble Sort without swaps are assumed slowest, with a runtime estimated of Ω(N^2).

Then, the worst case scenario of each sorting algorithm is hypothesized. Mergesort will be the fastest with an estimated runtime of O(N LOG N). Each of the other five sorting algorithms all have an estimated runtime of O(N^2) which makes a hypothesis more difficult in this case. Bubblesort is notoriously slow and expected to rank slowest of the bunch.

Hypothesizing runtime for the average case scenario arrays combines reasoning from the previous cases. Quicksort and Mergesort are expected to be the fastest with a runtime of Θ(N LOG N). The next slowest then being Bubble Sort with swaps, Bubble Sort without swaps Insertion Sort and Selection Sort with an estimated runtime of Θ(N^2). Again, both of the Bubble Sorts are assumed to be slow.

After looking at the data from running these nine arrays with each of the six sorting algorithms, the data line up well. The results for the arrays of length 100 can be generalized easily. Slight distinctions can be seen between all of the values, but are negligible as times are differences in the microsecond to millisecond range.

For the arrays of size 1,000, the timings are more distinguished. In the best case category of arrays size 1,000, Insertion Sort and Bubble Sort with swaps were the fastest, with linear order of growth, ranking in the microseconds. Next slowest, ranking in the milliseconds was Mergesort and Quicksort, which which both logarithmic order of growth. Finally, the slowest was Selection Sort and Bubble Sort without swaps, with exponential order of growth. Both algorithms had times that were in the hundreths of seconds, as expected. These patterns, as expected, can be seen throughout the worst case and average case scenarios.

The most noticeable difference in array cases of size 10,000. For the best case arrays, Insertion Sort and Bubble Sort with swaps can really be noticed. These algorithms do not do comparisons for sorted portions and their time shows so, ranking in the miroseconds. Quicksort and Mergesort then follow close there after in the hundredths of seconds. Lastly for the best case, the exponential algorithms, Selection Sort and Bubble Sort without swaps both took multiple seconds to complete.

The worst case and average case both followed the hypothesizes earlier stated. The timings of Quicksort were difficult to compare for the array of size 1,000 or 10,000 because of the size of recursion allowed for the time to finish. Timings in the data provided was for a Quicksort Algorithm that ran successfully but did not provide results that corresponded to the hypothesis.

All other data held strong to the hypothesizes. Bubbles Sort showed true that it is the slug of the bunch. Mergesort and Quicksort, were the most efficient in their worst cases compared to the other four algorithms. This project was a great way to understand order of growth and will be referenced in the future.

Graphics:

Selection Sort:

Best T(N) = Ω(N^2)
Worst T(N) = O(N^2)
Average T(N) = Θ(N^2)

Best	Worst	Average
.003	.003	.003
.03	.03	.03
3.25	3.14	3.07

Insertion Sort:

Best T(N) = Ω(N)
Worst T(N) = O(N^2)
Average T(N) = Θ(N^2)

Best	Worst	Average
.00001	.0003	.00004
.0001	.039	.005
.001	4.06	.53

Bubble Sort w/ Swap:

Best T(N) = Ω(N)
Worst T(N) = O(N^2)
Average T(N) = Θ(N^2)

Best	Worst	Average
.00001	.0005	.000029
.00007	.069	.042
.0008	7.3	4.2

Bubble Sort w/o Swap:

Best T(N) = Ω(N^2)

Worst T(N) = O(N^2)
Average T(N) = Θ(N^2)

Best	Worst	Average
.0004	.0005	.00038
.035	.062	.038
3.6	6.5	4.1

Quicksort:

Best T(N) = Ω(N LOG N)
Worst T(N) = O(N^2)
Average T(N) = Θ(N LOG N)

Best	Worst	Average
.0002	.0002	.0002
.002	.002	.002
.027	.025	.026

Mergesort:

Best T(N) = Ω(N LOG N)
Worst T(N) = O(N LOG N)
Average T(N) = Θ(N LOG N)

Best	Worst	Average
.0002	.0002	.0002
.002	.003	.002
.029	.037	.036

Graph of Order of Operations: