Walker Mitchell
CS361
5/5/2019

Lab 2 - More Sorts

Task 1: Radix Sort

This task was to implement Radix Sort. I found the pseudocode in the book to be of no help whatsoever:

RADIX-SORT(A, d)
1 **for** $i = 1$ **to** d
2 use a stable sort to sort array A on digit i

So, I had to look elsewhere for reference material. I ended up using my notes from class in conjunction with https://www.geeksforgeeks.org/radix-sort/.

```
# Task 1: Radix Sort
# The book's pseudocode is totally useless. I'm using https://www.geeksforgeeks.org/radix-sort/ as my source.
def radixSort(arr):
    digit = 1
    biggest = max(arr)
    while(digit < biggest):
        countingSort(arr, digit)
        digit *= 10

# Counting Sort logic based on the above geeksforgeeks source.
# I can't use Quicksort here because Radix needs a stable secondary sort.
def countingSort(arr, digit):
    length = len(arr)
    sorted = [0] * length       # Initialize output array
    count = [0] * 10            # Initialize counter array

    for i in range(0, length):  # Fill counter array
        index = arr[i]//digit
        count[index%10] += 1

    for i in range(1, 10):      # Set counter array to be filled with relevant indices
        count[i] += count[i-1]

    j = length - 1
    while j >= 0:               # Fill output array
        index = arr[j]//digit
        sorted[count[index%10] -1] = arr[j]
        count[index%10] -= 1
        j -= 1

    for i in range(0,length):   # Overwrite input array with sorted output array
        arr[i] = sorted[i]
```

The second part of this task was to run this sort on an array of roughly 10,000,000 numbers using a provided data file. I used the file loading logic from Lab 1 to accomplish this, as Task 6 directs:

```
# Initialize data array
data = []

# Open the data file
file = open("lab2_data.txt", "r")

# Iterate through the file line by line, and fill the array with the values from the file.
for line in file:
    if line != "\n":
        data.append(int(line.strip("n\'")))

# Close the file now that we're done with it
file.close()
```

Task 2: Bucket Sort

This task was to implement Bucket Sort. For this I used my class notes as well as the book's pseudocode. There was initially some confusion on how to choose what the buckets were, but it got sorted out.

```
# Task 2: Bucket Sort
# Using the book's pseudocode.
def bucketSort(arr):
    outside = []
    inside = []
    lowest = min(arr)
    numBuckets = int(math.sqrt(max(arr)- lowest))
    for i in range(numBuckets):         # Set up buckets
        outside.append(inside)

    bigBucket = (numBuckets*numBuckets)+lowest
    for k in arr:           # Fill buckets
        if k >= (bigBucket):        # Handles cases where the else logic would place the element out of range
            outside[-1] = outside[-1] + [k]
        else:
            outside[(k-lowest)//numBuckets] = outside[(k-lowest)//numBuckets] + [k]

    #for i in range(numBuckets):        # Just checking that the buckets filled properly
    #    print(len(outside[i]))

    for i in range(numBuckets):         # Sort buckets
        auxQuickSort(outside[i], 0, len(outside[i])-1)

    sorted = []
    for i in range(numBuckets):         # Concatenate buckets into sorted list
        sorted += outside[i]
    return sorted
```

Task 3: Do the sorts work?

Task 3 was to test whether my Bucket and Radix sorts actually sort the data they're given. To do this I used my flgIsSorted function from Lab 1:

```
# I'm using the same logic from Lab 1 for checking whether the array is sorted.
# Since this code was proven to work in Lab 1, I can use it here for Task 3.
def flgIsSorted(arr):
    if len(arr) <= 1:                                  # When the array has only one
        return True

    if flgIsSorted(arr[:(len(arr)//2)]):                        # If the first half
        if flgIsSorted(arr[(len(arr)//2):]):                    # the second half i
            if arr[(len(arr)//2)] >= arr[(len(arr)//2)-1]:      # and the first ele
                return True                                     # the array is sort

    return False                                       # If the above conditions are
```

To test my Radix and Bucket sorts, I loaded the first ten elements of the data file then ran them separately on that data. I then used flgIsSorted to verify the validity of the sort, and printed the sorted data to demonstrate that everything is working properly:

```
5   testData = data[:10]
6   print(testData)
7
8   sortedData = bucketSort(testData)
9   print("Does my Bucket Sort work?")
0   print(flgIsSorted(sortedData))
1   print(sortedData)
2
3   radixSort(testData)
4   print("Does my Radix Sort work?")
5   print(flgIsSorted(testData))
6   print(testData)
```

```
ion file                                                    length : 4,838   lines : 1
[5449937, 7646474, 1202789, 3140323, 5406434, 6083164, 4088282, 9737449, 8323914, 4530583]
Does my Bucket Sort work?
True
[1202789, 3140323, 4088282, 4530583, 5406434, 5449937, 6083164, 7646474, 8323914, 9737449]
Does my Radix Sort work?
True
[1202789, 3140323, 4088282, 4530583, 5406434, 5449937, 6083164, 7646474, 8323914, 9737449]
```

Task 6: Finding the Top Ten

This task was to come up with a recursive algorithm that outputs the ten largest elements in an array, in descending order. The first way I thought of to do this was to use a sort then output the last 10 values, but Task 7 directs us to test our algorithm against this method so I surmised that we were intended to figure out another way. I initially used a greedy approach not dissimilar to iterating through the array one by one with a for loop. While it worked just fine with smaller arrays, when I tried applying it to an array of size 1000 it came back with a "recursion depth exceeded" error:

```
  File "Lab2.py", line 125, in findTopTen
    return findTopTen(arr, ten, (index+1))
  [Previous line repeated 96 more times]
  File "Lab2.py", line 129, in findTopTen
    return findTopTen(arr, ten[1:], (index+1))         # Continue recursion with the ten largest values in ten
  File "Lab2.py", line 125, in findTopTen
    return findTopTen(arr, ten, (index+1))
  File "Lab2.py", line 125, in findTopTen
    return findTopTen(arr, ten, (index+1))
  File "Lab2.py", line 125, in findTopTen
    return findTopTen(arr, ten, (index+1))
  [Previous line repeated 324 more times]
  File "Lab2.py", line 119, in findTopTen
    if index >= len(arr):              # If we've reached the end of the array then 'ten' contains the top ten values of t
he array.
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Since this error is designed to catch infinite recursion, and I knew I didn't have that sort of situation, I then tried increasing the maximum recursion depth using sys.setrecursionlimit(). With a larger recursion limit, however, my algorithm overflowed the stack:

```
Testing topTen
Traceback (most recent call last):
  File "Lab2.py", line 126, in findTopTen
    return findTopTen(arr, ten, (index+1))
  File "Lab2.py", line 126, in findTopTen
    return findTopTen(arr, ten, (index+1))
  File "Lab2.py", line 126, in findTopTen
    return findTopTen(arr, ten, (index+1))
  [Previous line repeated 997 more times]
MemoryError: Stack overflow
```

So I grudgingly admitted to myself that I needed to completely rethink my approach to this algorithm. I ended up using a divide-and-conquer algorithm inspired (somewhat) by Bucket Sort that:

1. Splits the n-element array into sqrt(n) pieces of size sqrt(n)
2. Quicksorts the pieces
3. Takes the 10 largest values from each piece
4. Concatenates the results into an array of size 10*sqrt(n)
5. Quicksorts that array
6. Outputs the 10 largest values from the result

```
# Task 6: Ten largest values
# Doing this one all by myself like a big boy.
def topTen(arr):
    if len(arr) <= 10:                  # If the array has ten values or less then just sort the array and we're done.
        auxQuickSort(arr, 0, len(arr)-1)
        return arr[::-1]                # Sort logic outputs in increasing order so we list it backwards.

    step = int(math.sqrt(len(arr)))
    temp = findTopTen(arr, 0, step)    # Returns a list of the 10 largest elements from each of the sqrt(len(arr)) pieces

    auxQuickSort(temp, 0, len(temp)-1)  # Sort the "candidates" for largest pieces
    temp = temp[len(temp)-10:len(temp)] # Take top ten
    return temp[::-1]                   # Return the top ten, backwards

def findTopTen(arr, index, step):
    if index >= step*step:                          # Stop when we run out of array
        auxQuickSort(arr, index, len(arr)-1)        # Sort what's left
        if (len(arr) - step*step) <= 10:            # If there are 10 or fewer elements left then return what we have
            return arr[step*step:len(arr)]
        else: return arr[len(arr)-10:len(arr)]       # Otherwise return largest 10
    else:
        auxQuickSort(arr, index, index+step-1)      # Quick Sort pieces
        return arr[index+step-10:index+step] + findTopTen(arr, index+step, step)    # Concatenate 10 largest with result of next level
```

For the quicksorting detailed above, I used my Lab 1 quicksort logic that is used for task 4:

```
# Quick Sort logic imported from Lab 1 for Task 4.
def auxQuickSort(arr, startIndex, endIndex):
    if startIndex < endIndex:
        partitionIndex = auxPartition(arr, startIndex, endIndex)
        auxQuickSort(arr, startIndex, partitionIndex)
        auxQuickSort(arr, (partitionIndex + 1), endIndex)

def auxPartition(arr, startIndex, endIndex):
    midPoint = (endIndex + startIndex)//2
    x = (arr[endIndex] + arr[startIndex] + arr[midPoint])//3
    i = startIndex-1
    for j in range(startIndex, endIndex+1):       # Pseudocode say
        if arr[j] <= x:                           # the pivot so w
            i += 1                                # the second par
            temp = arr[i]
            arr[i] = arr[j]
            arr[j] = temp
    return i
```

Tasks 4-8: Timing all the things

I've grouped these tasks together into one because that is how I accomplished the timing portion of them. Similar to Lab 1, the tasks (and portions of tasks) discussed here involved recording the runtime of this lab's algorithms on arrays with size starting at 1000 and increasing by a factor of 10 until they are run across the entire data file.

```
print("Now we're going to compare how long it takes Radix Sort vs. Bucket Sort vs. Quick Sort to sort arrays of increasing size.")
print("Then we will compare my topTen algorithm against sorting with QuickSort then manually printing the ten largest values.")
nums = len(data)
#sys.setrecursionlimit(nums)
arraySize = 1000
while arraySize <= nums:
    print("Starting with Radix Sort, sorting with size " + str(arraySize))
    radixData = data[:arraySize]
    timer = time.time()
    radixSort(radixData)
    timer = (time.time() - timer) * 1000
    print("Radix Sort took " + str(timer) + " milliseconds.")

    print("Did it sort correctly?")
    print(flgIsSorted(radixData))

    print("Now let's try using Bucket Sort.")

    bucketData = data[:arraySize]
    timer = time.time()
    bucketData = bucketSort(bucketData)
    timer = (time.time() - timer) * 1000
    print("Bucket Sort took " + str(timer) + " milliseconds.")

    print("Did it sort correctly?")
    print(flgIsSorted(bucketData))

    print("Now let's compare those against Quick Sort.")

    quickData = data[:arraySize]
    timer = time.time()
    auxQuickSort(quickData, 0, len(quickData)-1)
    timer = (time.time() - timer) * 1000
    print("Quick Sort took " + str(timer) + " milliseconds.")
```

```
    quickData = data[:arraySize]
    timer = time.time()
    auxQuickSort(quickData, 0, len(quickData)-1)
    timer = (time.time() - timer) * 1000
    print("Quick Sort took " + str(timer) + " milliseconds.")

    print("Now let's find the top ten.")
    testData = data[:arraySize]

    print("Testing topTen")
    timer = time.time()
    print(topTen(testData))
    timer = (time.time() - timer) * 1000
    print("topTen took " + str(timer) + " milliseconds.")

    print("Compare to:")
    timer = time.time()
    auxQuickSort(testData, 0, len(testData)-1)
    testData.reverse()
    timer = (time.time() - timer) * 1000
    print(testData[:10])
    print("Quick Sorting to find top ten took " + str(timer) + " milliseconds.")
    print("")

    arraySize *= 10
```

The file doesn't contain exactly 10,000,000 numbers, so the final iteration had to be hardcoded.

```
# The given data file has 9,999,097 numbers in it instead of 10,000,000
# so I'm doing the final iteration across the entire file manually
print("Starting with Radix Sort, sorting with size 9,999,097")
radixData = data
timer = time.time()
radixSort(radixData)
timer = (time.time() - timer) * 1000
print("Radix Sort took " + str(timer) + " milliseconds.")

print("Did it sort correctly?")
print(flgIsSorted(radixData))

print("Now let's try using Bucket Sort.")

bucketData = data
timer = time.time()
bucketData = bucketSort(bucketData)
timer = (time.time() - timer) * 1000
print("Bucket Sort took " + str(timer) + " milliseconds.")

print("Did it sort correctly?")
print(flgIsSorted(bucketData))

print("Now let's compare those against Quick Sort.")
```

```
quickData = data
timer = time.time()
auxQuickSort(quickData, 0, len(quickData)-1)
timer = (time.time() - timer) * 1000
print("Quick Sort took " + str(timer) + " milliseconds.")
print("")

print("Now let's find the top ten.")
testData = data

print("Testing topTen")
timer = time.time()
print(topTen(testData))
timer = (time.time() - timer) * 1000
print("topTen took " + str(timer) + " milliseconds.")

print("Compare to:")
timer = time.time()
auxQuickSort(testData, 0, len(testData)-1)
testData.reverse()
timer = (time.time() - timer) * 1000
print(testData[:10])
print("Quick Sorting to find top ten took " + str(timer) + " milliseconds.")
```

This (my final code) output the following:

```
[9999879, 9999791, 9999787, 9999620, 9999123, 9999011, 9998977, 9998883, 9998858, 9998730]
Quick Sorting to find top ten took 407.9585075378418 milliseconds.

Starting with Radix Sort, sorting with size 1000000
Radix Sort took 6760.932683944702 milliseconds.
Did it sort correctly?
True
Now let's try using Bucket Sort.
Bucket Sort took 4214.736223220825 milliseconds.
Did it sort correctly?
True
Now let's compare those against Quick Sort.
Quick Sort took 4972.708702087402 milliseconds.
Now let's find the top ten.
Testing topTen
[9999994, 9999982, 9999977, 9999971, 9999900, 9999894, 9999882, 9999879, 9999867, 9999865]
topTen took 3053.8530349731445 milliseconds.
Compare to:
[9999994, 9999982, 9999977, 9999971, 9999900, 9999894, 9999882, 9999879, 9999867, 9999865]
Quick Sorting to find top ten took 5195.16396522522 milliseconds.

Starting with Radix Sort, sorting with size 9,999,097
Radix Sort took 75357.56659507751 milliseconds.
Did it sort correctly?
True
Now let's try using Bucket Sort.
Bucket Sort took 135019.12927627563 milliseconds.
Did it sort correctly?
True
Now let's compare those against Quick Sort.
Quick Sort took 57184.12709236145 milliseconds.

Now let's find the top ten.
Testing topTen
[9999999, 9999998, 9999997, 9999996, 9999995, 9999994, 9999993, 9999992, 9999991, 9999990]
topTen took 33299.072265625 milliseconds.
Compare to:
[9999999, 9999998, 9999997, 9999996, 9999995, 9999994, 9999993, 9999992, 9999991, 9999990]
Quick Sorting to find top ten took 58677.22582817078 milliseconds.
```

I then assembled the data from running my code 3 times to form the following tables:

		Run 1 Runtime		
		(in milliseconds)		
Array Size	Radix Sort	Bucket Sort	Quick Sort	topTen
1000	5	52.9	2	2
10000	55.8	67.8	32.9	24.9
100000	563.5	225.4	410.9	268.3
1000000	6636.3	4044.2	4937.8	3083.7
9999097	73562.4	134209.4	56987.7	32825.3

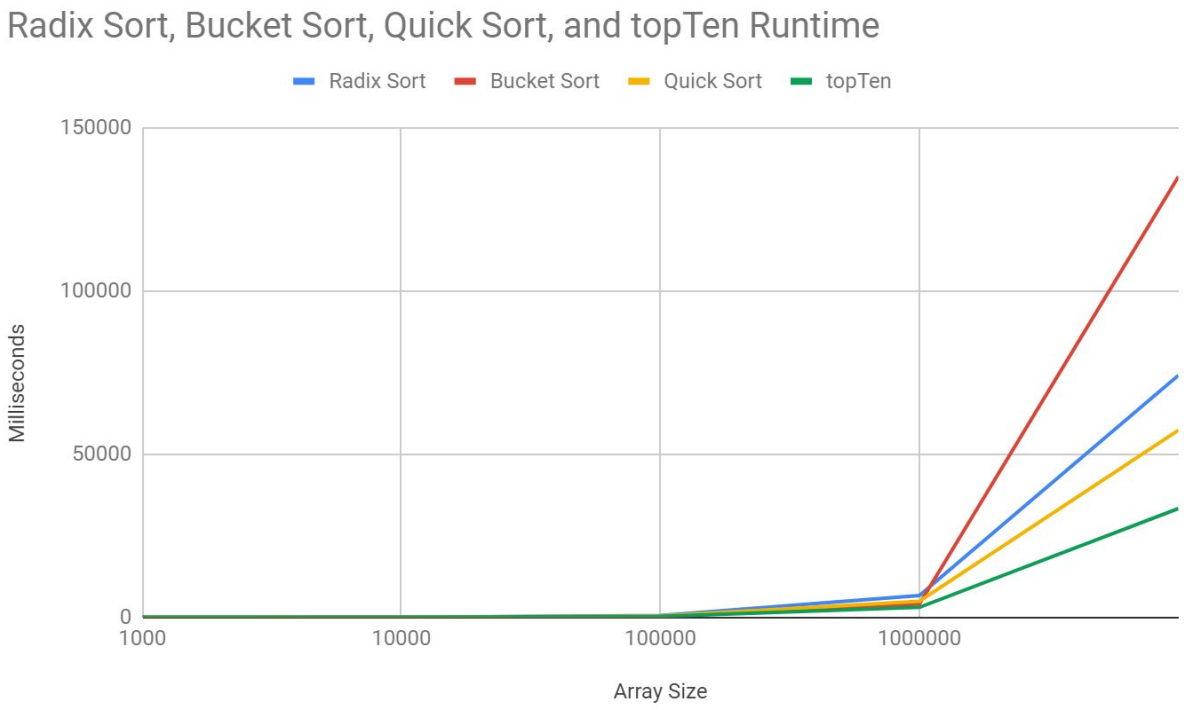
		Run 2 Runtime		
		(in milliseconds)		
Array Size	Radix Sort	Bucket Sort	Quick Sort	topTen
1000	5	52.9	3	3
10000	56.9	68.9	32.9	25.9
100000	565.5	224.4	413.9	267.3
1000000	6694.1	4054.2	4895.9	3101.8
9999097	73713	136184	57962.1	34032.1

		Run 3 Runtime		
		(in milliseconds)		
Array Size	Radix Sort	Bucket Sort	Quick Sort	topTen
1000	5	53.9	3	3
10000	57.9	67.8	33.9	27.9
100000	572.5	230.4	413.9	268.3
1000000	6760.9	4214.7	4972.7	3053.9
9999097	75357.6	135019.1	57184.1	33299.1

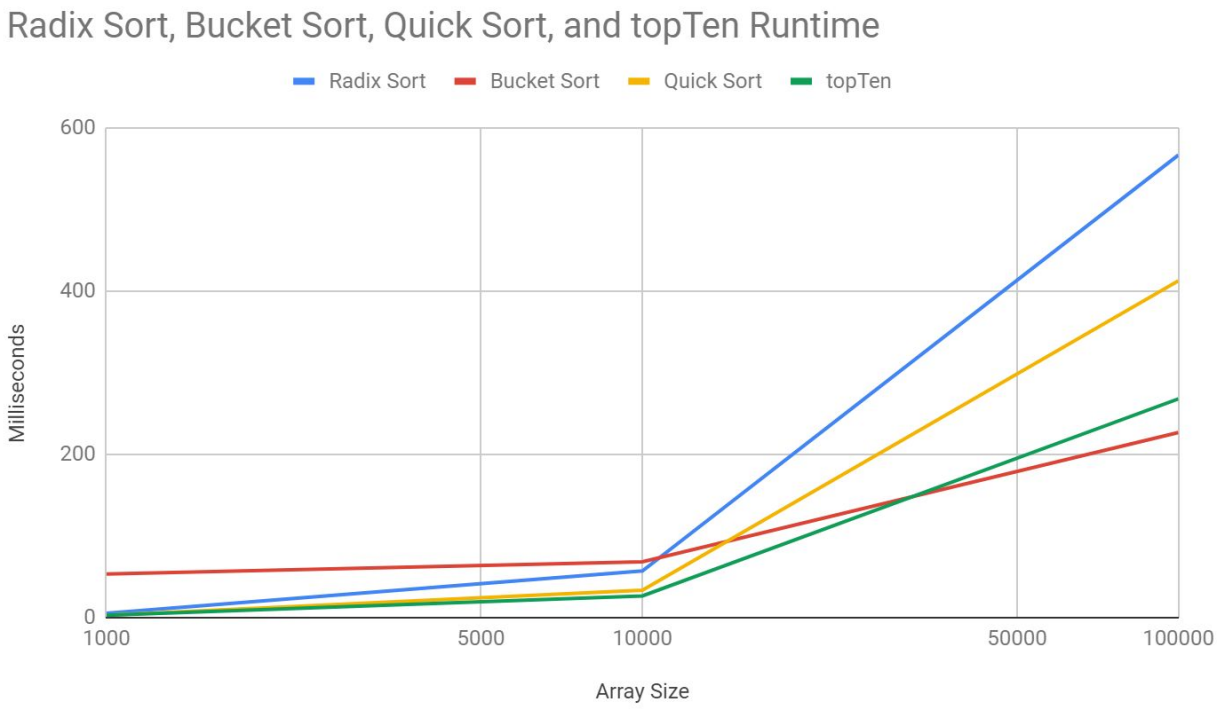
Using these tables, I constructed the following table of averages:

		Average Runtime		
		(in milliseconds)		
Array Size	Radix Sort	Bucket Sort	Quick Sort	topTen
1000	5	53.23333333	2.666666667	2.666666667
10000	56.86666667	68.16666667	33.23333333	26.23333333
100000	567.1666667	226.7333333	412.9	267.9666667
1000000	6697.1	4104.366667	4935.466667	3079.8
9999097	74211	135137.5	57377.96667	33385.5

Using this table of averages I constructed the following graph:



This is using a logarithmic scale on the horizontal axis to represent that the array size is being multiplied by 10 each iteration. We run into the same problem here as we did in Lab 1, which is that we can't really see the results for the 3 smallest sizes at this scale. So here's a graph that's zoomed in so we can see those results:



These graphs show a rather unexpected result: on average, Quicksort is actually running faster than either Radix Sort or Bucket Sort! While Bucket Sort was the fastest sort on sizes 100,000 and 1,000,000, Radix Sort was not faster at any point than Quicksort. I take this to signify that my implementation of Radix and Bucket Sort left something to be desired. While I'm not sure exactly what went wrong with Radix Sort, I would guess that my problem with Bucket Sort lies in how I choose my buckets. Specifically, I think I don't have enough, causing each bucket to be filled with more elements than is optimal.