Ian G (Mon / Wed)

Evan LaFleur (Tues / Thurs)

Michael Robertson (Tues, Thurs)

QuickSort Algorithm Test Report

CS350

21st Oct, 2021

Abstract:

This report tested the efficiency of a quicksort algorithm implemented in python 3.10 and ran on two machines. The performance provided by these two machines was similar enough that for the purposes of this report the results from them can be combined. The performance results of these tests seem to indicate a big O value of O(n log(n)), meaning that it performed much better than the assumed worst case of O(n^2) in reality. The below report will further elaborate on the workings of the specific implementation of the algorithm as well as explain the raw data displayed.

Introduction:

This report will be divided into three sections beyond this introduction. The background will be split into a section discussing what quicksort is and the difference between the two machines used to test the sorting algorithm. The analysis procedure and results section will walk through the code and discuss the results. Lastly, the appendix section will include the graphs produced by the python program which ran the quicksort analysis.

Background:

What is Quicksort?

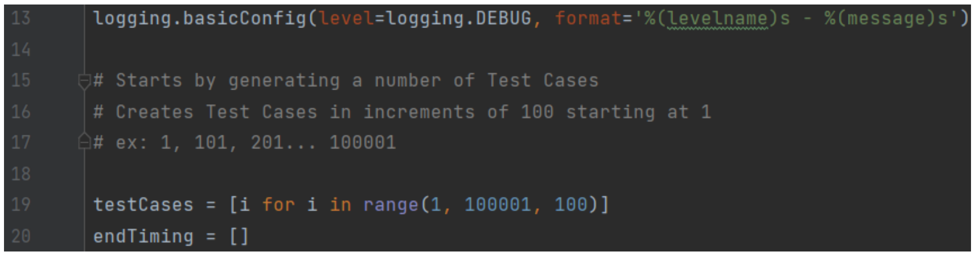
Quicksort is an in-place divide-and-conquer sorting algorithm designed first by Tony Hoare in 1959. The process for the sort can be split into four distinct steps. First, check to see if the range the algorithm is looking at is less than two values and return if so. Second, if the first part is not the case, then pick a value (a pivot) that appears in the range – usually the last one. Third, the algorithm will partition the range by moving the chosen pivot down the array until it is equal to the point of division (where it should be in its final position). Fourth and last, recurse into the sub-ranges created by the partitions (which ignores the pivot). Repeat until sorted. A more detailed analysis will be provided with the code in the Analysis section of this report.

Machine Differences?

For the program, which was written to test this algorithm, two machines were used. The first was a virtual Linux machine in the form of Babbage, and the second was a MacBook Pro. While these two machines are fundamentally different, because of the single core design of the program, a discernable difference was not noticed in the execution of the program (results appear to be within 10% of each other). If the program was rewritten to optimize for multiple cores, the difference would in theory be much greater due to Babbage's significantly higher core count (it must be said though, that due to the virtual nature of the environment, this may not be the case).

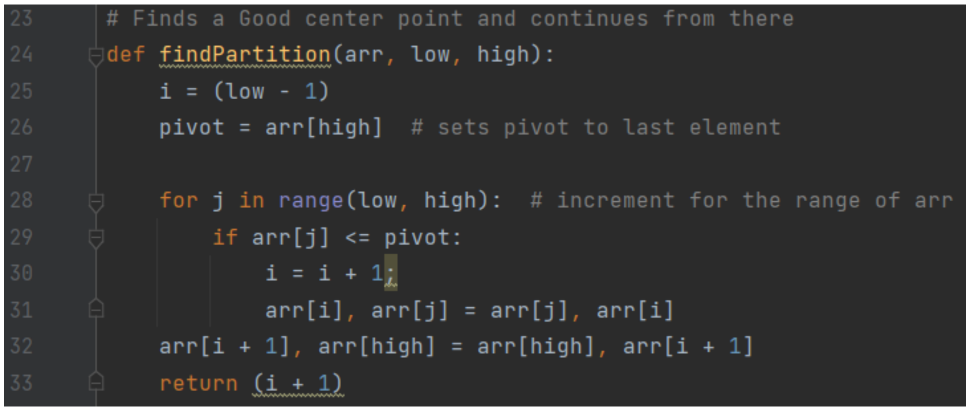
Analysis Procedure and Results:

Based off the raw data graphs in the appendix, an estimated function of $\frac{x(\log x)}{1500000}$ where x is the number of items to sort and the output is the time in seconds appears to accurately represent the efficiency of the implementation of quicksort. Speaking of the quicksort implementation, it goes as follows:



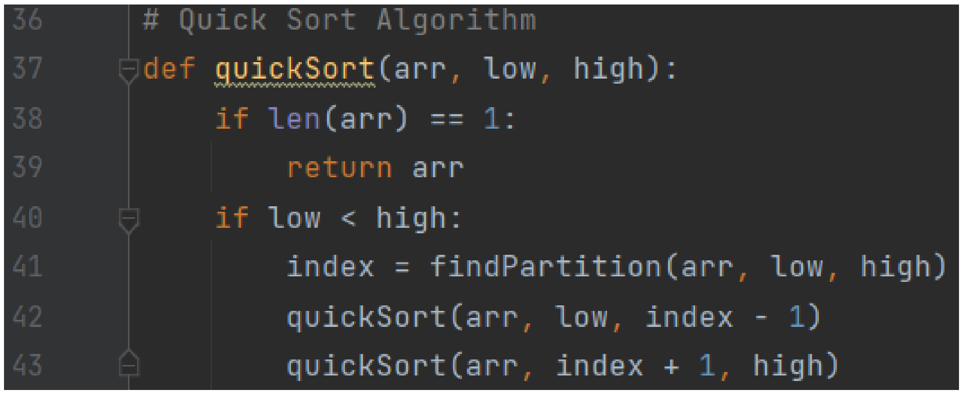
```python
logging.basicConfig(level=logging.DEBUG, format='%(levelname)s - %(message)s')

# Starts by generating a number of Test Cases
# Creates Test Cases in increments of 100 starting at 1
# ex: 1, 101, 201... 100001

testCases = [i for i in range(1, 100001, 100)]
endTiming = []
```

Starting off here (after inclusions), we have the setup of the logging and the generation of the number of test cases – as well as the creation of an array to store the timing.



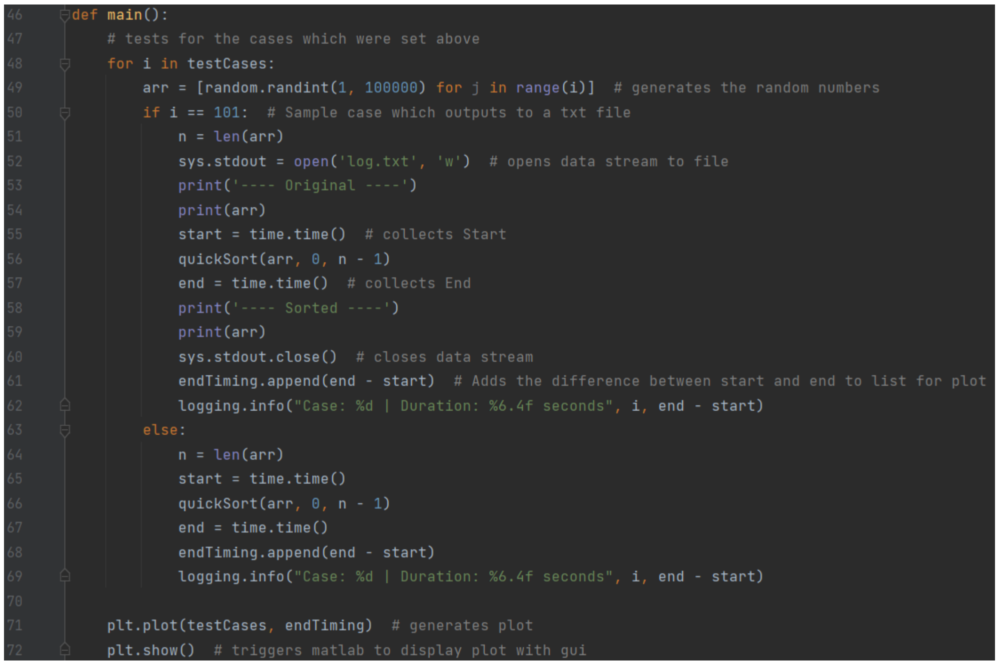
```python
# Finds a Good center point and continues from there
def findPartition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]  # sets pivot to last element

    for j in range(low, high):  # increment for the range of arr
        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)
```

Next, we have the function which does the partition and returns the point in the starting array which actively partitioned the data. In this version of quicksort, the last value in the array was chosen as the pivot.



```python
# Quick Sort Algorithm
def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        index = findPartition(arr, low, high)
        quickSort(arr, low, index - 1)
        quickSort(arr, index + 1, high)
```

Here, we have the actual implementation of the quicksort algorithm, as described in the background.



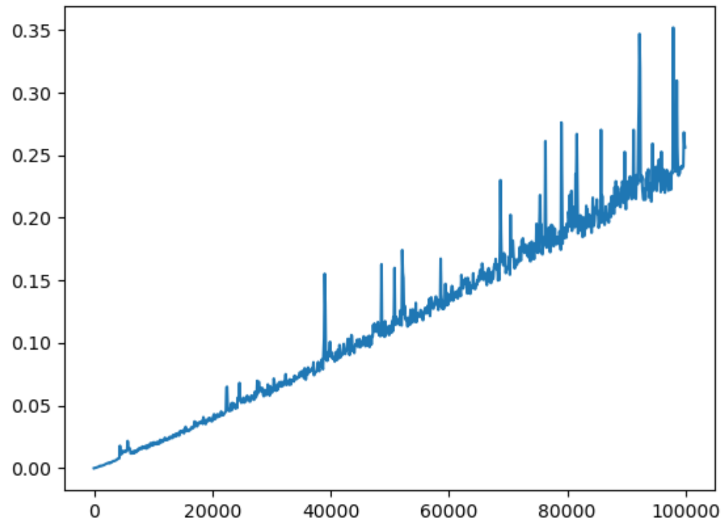
```python
def main():
    # tests for the cases which were set above
    for i in testCases:
        arr = [random.randint(1, 100000) for j in range(i)]  # generates the random numbers
        if i == 101:  # Sample case which outputs to a txt file
            n = len(arr)
            sys.stdout = open('log.txt', 'w')  # opens data stream to file
            print('---- Original ----')
            print(arr)
            start = time.time()  # collects Start
            quickSort(arr, 0, n - 1)
            end = time.time()  # collects End
            print('---- Sorted ----')
            print(arr)
            sys.stdout.close()  # closes data stream
            endTiming.append(end - start)  # Adds the difference between start and end to list for plot
            logging.info("Case: %d | Duration: %6.4f seconds", i, end - start)
        else:
            n = len(arr)
            start = time.time()
            quickSort(arr, 0, n - 1)
            end = time.time()
            endTiming.append(end - start)
            logging.info("Case: %d | Duration: %6.4f seconds", i, end - start)

    plt.plot(testCases, endTiming)  # generates plot
    plt.show()  # triggers matlab to display plot with gui
```

Lastly, for posterity, we have the main function, which simply calls the quicksort tool repeated until the desired number of test cases are ran. Note that this also creates the plots which will be seen below in the appendix.

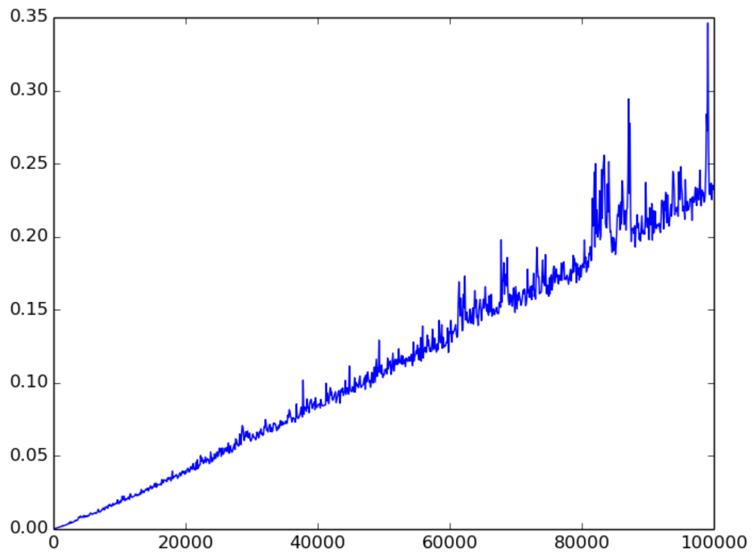
Appendix:

Note: for all three appendix items below, the x axis is number of items sorted and the y axis is amount of time spent sorting that number of items. The appearance difference of item one vs items two and three is due to it being the one ran on Babbage, while two and three were ran on the MacBook Pro.

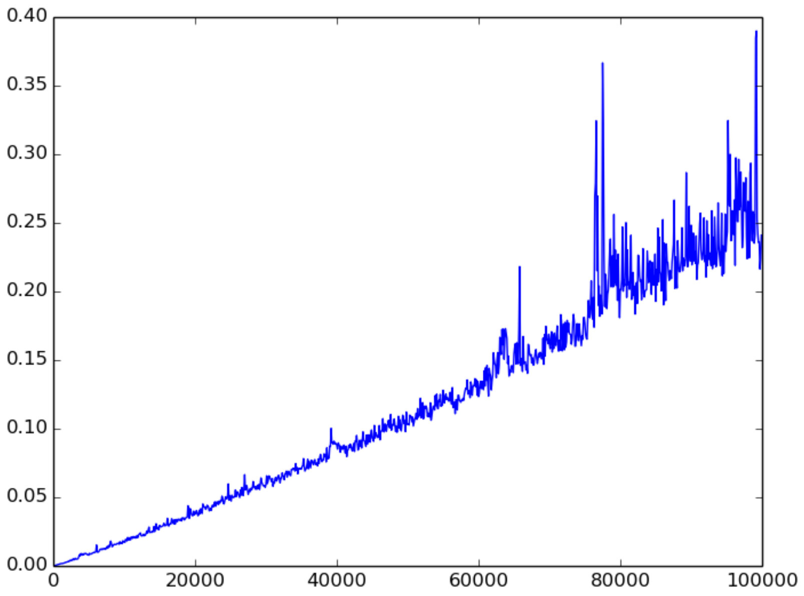
Item One



Item Two



Item Three


0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20000
40000
60000
80000
100000