Final Paper

An Investigation into Multi-Pivot Quicksort Algorithms

Authors:
Paymahn Moghadasian [umpaymah@myumanitoba.ca]

Joshua Hernandez [umhern23@myumanitoba.ca]

Instructor: Dr. S. Durocher

COMP 4420
Advanced Design and Analysis of Algorithms

Due : April 9, 2014

Contents

Sort Method	Comparisons
Classic	$2n \log n - 1.51n + O(\log(n))$
Dual Pivot	$2.13n \log n - 2.57n + O(\log(n))$
Optimal Dual Pivot	$1.8n \log n + O(n)$
Three Pivot	$1.846n \log n + O(n)$
Yaroslavskiy	$1.9n \log n - 2.46n + O(\log(n))$
M Pivot	$O(n \log n)$

Table 1: Summary table of theoretical comparisons.

1 Introduction

Test

2 Quicksort

2.1 Classic Quicksort

2.2 Dual Pivot Quicksort

2.3 Optimal Dual Pivot Quicksort

2.4 Three Pivot Quicksort

2.5 Yaroslavskiy Quicksort

2.6 M Pivot Quicksort

RAWR

2.6.1 Testing

Just to test the subsection code. Test text : Recall from section 2.1 on page 1

2.7 Summary

3 Analysis

Test

Sort Method	Swaps
Classic	$0.33n \log n - 0.58n + O(\log(n))$
Dual Pivot	$0.8n \log n - 0.3n + O(\log(n))$
Optimal Dual Pivot	$0.33n \log n + O(n)$
Three Pivot	$0.615n \log n + O(n)$
Yaroslavskiy	$0.6n \log n + 0.08n + O(\log(n))$
M Pivot	$O(n \log n)$

Table 2: Summary table of theoretical swaps.

3.1 Data Collection

3.2 Data Processing

3.3 Non-Linear Curve Fit

3.4 Summary

Sort Method	$A_{comparisons}$
ClassicQuicksort - 1 - 1	0.02219 ± 0.00045
ClassicQuicksort - 2 - 1	0.02126 ± 0.00015
ClassicQuicksort - 3 - 1	0.01799 ± 0.00008
DualPivotQuicksort - 1 - 2	0.02109 ± 0.00024
DualPivotQuicksort - 2 - 2	0.01787 ± 0.00008
HeapOptimizedMPivotQuicksort - 1 - 3	0.02755 ± 0.00027
HeapOptimizedMPivotQuicksort - 1 - 4	0.02782 ± 0.00019
HeapOptimizedMPivotQuicksort - 1 - 5	0.02903 ± 0.00029
HeapOptimizedMPivotQuicksort - 1 - 6	0.02801 ± 0.00019
MPivotQuicksort - 1 - 3	0.01955 ± 0.00010
MPivotQuicksort - 1 - 4	0.02039 ± 0.00013
MPivotQuicksort - 1 - 5	0.02136 ± 0.00013
MPivotQuicksort - 1 - 6	0.02369 ± 0.00012
OptimalDualPivotQuicksort - 1 - 2	0.02044 ± 0.00023
OptimalDualPivotQuicksort - 2 - 2	0.01754 ± 0.00008
ThreePivotQuicksort - 1 - 3	0.02595 ± 0.00008
YaroslavskiyQuicksort - 1 - 2	0.01811 ± 0.00009

Table 3: Summary table coefficients of the non-linear fit for the parameter A on the comparison data.

Sort Method	$B_{comparisons}$
ClassicQuicksort - 1 - 1	-0.06733 ± 0.01137
ClassicQuicksort - 2 - 1	-0.04391 ± 0.00382
ClassicQuicksort - 3 - 1	-0.01913 ± 0.00195
DualPivotQuicksort - 1 - 2	-0.05991 ± 0.00618
DualPivotQuicksort - 2 - 2	-0.01424 ± 0.00194
HeapOptimizedMPivotQuicksort - 1 - 3	-0.04476 ± 0.00672
HeapOptimizedMPivotQuicksort - 1 - 4	-0.05037 ± 0.00480
HeapOptimizedMPivotQuicksort - 1 - 5	-0.06169 ± 0.00741
HeapOptimizedMPivotQuicksort - 1 - 6	-0.02207 ± 0.00470
MPivotQuicksort - 1 - 3	-0.01587 ± 0.00241
MPivotQuicksort - 1 - 4	-0.00828 ± 0.00333
MPivotQuicksort - 1 - 5	-0.00157 ± 0.00327
MPivotQuicksort - 1 - 6	-0.02795 ± 0.00296
OptimalDualPivotQuicksort - 1 - 2	-0.05680 ± 0.00589
OptimalDualPivotQuicksort - 2 - 2	-0.01536 ± 0.00212
ThreePivotQuicksort - 1 - 3	-0.04484 ± 0.00196
YaroslavskiyQuicksort - 1 - 2	-0.01998 ± 0.00236

Table 4: Summary table coefficients of the non-linear fit for the parameter B on the comparison data.

Sort Method	$C_{comparisons}$
ClassicQuicksort - 1 - 1	249.54532 ± 242.31658
ClassicQuicksort - 2 - 1	37.44145 ± 81.34333
ClassicQuicksort - 3 - 1	5.72417 ± 41.62533
DualPivotQuicksort - 1 - 2	212.91009 ± 131.86339
DualPivotQuicksort - 2 - 2	-24.95784 ± 41.40114
HeapOptimizedMPivotQuicksort - 1 - 3	55.15044 ± 143.28120
HeapOptimizedMPivotQuicksort - 1 - 4	145.67625 ± 102.42020
HeapOptimizedMPivotQuicksort - 1 - 5	173.26676 ± 157.95242
HeapOptimizedMPivotQuicksort - 1 - 6	-96.99056 ± 100.14751
MPivotQuicksort - 1 - 3	44.96143 ± 51.37619
MPivotQuicksort - 1 - 4	8.05960 ± 70.95161
MPivotQuicksort - 1 - 5	-41.33638 ± 69.78081
MPivotQuicksort - 1 - 6	104.54914 ± 63.10871
OptimalDualPivotQuicksort - 1 - 2	200.81644 ± 125.58142
OptimalDualPivotQuicksort - 2 - 2	-5.62515 ± 45.13255
ThreePivotQuicksort - 1 - 3	-3.12267 ± 41.72249
YaroslavskiyQuicksort - 1 - 2	15.80827 ± 50.32376

Table 5: Summary table coefficients of the non-linear fit for the parameter C on the comparison data.

Sort Method	A_{swap}
ClassicQuicksort - 1 - 1	0.01060 ± 0.00029
ClassicQuicksort - 2 - 1	0.01110 ± 0.00022
ClassicQuicksort - 3 - 1	0.00828 ± 0.00020
DualPivotQuicksort - 1 - 2	0.00636 ± 0.00016
DualPivotQuicksort - 2 - 2	0.00603 ± 0.00010
HeapOptimizedMPivotQuicksort - 1 - 3	0.00999 ± 0.00014
HeapOptimizedMPivotQuicksort - 1 - 4	0.00885 ± 0.00004
HeapOptimizedMPivotQuicksort - 1 - 5	0.00809 ± 0.00006
HeapOptimizedMPivotQuicksort - 1 - 6	0.00769 ± 0.00008
MPivotQuicksort - 1 - 3	0.00640 ± 0.00007
MPivotQuicksort - 1 - 4	0.00594 ± 0.00007
MPivotQuicksort - 1 - 5	0.00532 ± 0.00003
MPivotQuicksort - 1 - 6	0.00524 ± 0.00003
OptimalDualPivotQuicksort - 1 - 2	0.00636 ± 0.00016
OptimalDualPivotQuicksort - 2 - 2	0.00603 ± 0.00010
ThreePivotQuicksort - 1 - 3	0.00616 ± 0.00008
YaroslavskiyQuicksort - 1 - 2	0.00584 ± 0.00007

Table 6: Summary table coefficients of the non-linear fit for the parameter A on the swap data.

Sort Method	B_{swap}
ClassicQuicksort - 1 - 1	-0.00988 ± 0.00728
ClassicQuicksort - 2 - 1	-0.01734 ± 0.00552
ClassicQuicksort - 3 - 1	0.02413 ± 0.00496
DualPivotQuicksort - 1 - 2	0.00661 ± 0.00412
DualPivotQuicksort - 2 - 2	0.01175 ± 0.00251
HeapOptimizedMPivotQuicksort - 1 - 3	0.00893 ± 0.00364
HeapOptimizedMPivotQuicksort - 1 - 4	0.01414 ± 0.00099
HeapOptimizedMPivotQuicksort - 1 - 5	0.01868 ± 0.00163
HeapOptimizedMPivotQuicksort - 1 - 6	0.02127 ± 0.00193
MPivotQuicksort - 1 - 3	0.01946 ± 0.00166
MPivotQuicksort - 1 - 4	0.02154 ± 0.00173
MPivotQuicksort - 1 - 5	0.03241 ± 0.00079
MPivotQuicksort - 1 - 6	0.03319 ± 0.00070
OptimalDualPivotQuicksort - 1 - 2	0.00661 ± 0.00412
OptimalDualPivotQuicksort - 2 - 2	0.01175 ± 0.00251
ThreePivotQuicksort - 1 - 3	0.01487 ± 0.00204
YaroslavskiyQuicksort - 1 - 2	0.01614 ± 0.00189

Table 7: Summary table coefficients of the non-linear fit for the parameter B on the swap data.

Sort Method	C_{swap}
ClassicQuicksort - 1 - 1	18.65661 ± 155.26896
ClassicQuicksort - 2 - 1	105.27050 ± 117.70706
ClassicQuicksort - 3 - 1	-141.53414 ± 105.67454
DualPivotQuicksort - 1 - 2	-35.76973 ± 87.76549
DualPivotQuicksort - 2 - 2	32.79541 ± 53.55164
HeapOptimizedMPivotQuicksort - 1 - 3	-78.92155 ± 77.56035
HeapOptimizedMPivotQuicksort - 1 - 4	-16.98220 ± 21.20603
HeapOptimizedMPivotQuicksort - 1 - 5	-12.21900 ± 34.74970
HeapOptimizedMPivotQuicksort - 1 - 6	-11.41928 ± 41.05615
MPivotQuicksort - 1 - 3	-12.91440 ± 35.31595
MPivotQuicksort - 1 - 4	9.63570 ± 36.94081
MPivotQuicksort - 1 - 5	-19.44025 ± 16.74541
MPivotQuicksort - 1 - 6	18.13305 ± 14.95520
OptimalDualPivotQuicksort - 1 - 2	-35.76973 ± 87.76549
OptimalDualPivotQuicksort - 2 - 2	32.79541 ± 53.55164
ThreePivotQuicksort - 1 - 3	-8.94031 ± 43.54268
YaroslavskiyQuicksort - 1 - 2	6.43541 ± 40.21087

Table 8: Summary table coefficients of the non-linear fit for the parameter C on the swap data.

References

A Quicksort Code

A.1 Base Quicksort

```
__author__ = 'paymahnmoghadasian'


class BaseQuicksort():
    '''
    Base class for the quicksorts that we will implement
    '''
    def __init__(self, data, doInsertionSort = False, insertionSortThreshold=13,
         pivotSelection=1, numPivots=1):
        self.__numSwaps = 0
        self.__numComparisons = 0
        self.__data = data
        self.__doInsertionSort = doInsertionSort
        self.__insertionSortThreshold = insertionSortThreshold
        self.__pivotSelection = pivotSelection
        self.__numPivots = numPivots

    def __getPivotSelection(self):
        return self.__pivotSelection
    pivotSelection = property(__getPivotSelection)

    def __getNumPivots(self):
        return self.__numPivots
    numPivots = property(__getNumPivots)

    def __getInsertionSortThreshold(self):
        return self.__insertionSortThreshold
    insertionSortThreshold = property(__getInsertionSortThreshold)

    def __getDoInsertionSort(self):
        return self.__doInsertionSort
    doInsertionSort = property(__getDoInsertionSort)

    def getNumSwaps(self):
        return self.__numSwaps
    def setNumSwaps(self, value):
        self.__numSwaps = value
    def delNumSwaps(self):
```

```
        del self.__numSwaps
    numSwaps = property(getNumSwaps, setNumSwaps, delNumSwaps, "The number of
        swaps done by the sort algorithm")

    def getNumComparisons(self):
        return self.__numComparisons
    def setNumComparisons(self, value):
        self.__numComparisons = value
    def delNumComparisons(self):
        del self.__numComparisons
    numComparisons = property(getNumComparisons, setNumComparisons,
        delNumComparisons, "The number of comparisons done by the sorting
        algorithm")

    def getData(self):
        return self.__data
    def setData(self, value):
        self.__data = value
    def delData(self):
        del self.__data
    data = property(getData, setData,delData, "The data to be sorted")


    def sort(self):
        raise NotImplementedError("Cannot sort on base quicksort")

    def _insertionSort(self, data, lower, upper):
        '''
        Sorts self.data from [lower,upper) using insertion sort
        Lower is an inclusive bound
        Upper is an exclusive bound
        '''
        for i in xrange(lower + 1, upper):
            j = i
            while j > lower and self.lessThan(data[j],  data[j-1]):
                data[j], data[j-1] = data[j-1], data[j]
                self.numSwaps += 1
                j -= 1

        return data

    def lessThan(self, a, b):
```

```
        '''
        determines whether a < b
        '''
        self.numComparisons += 1
        return a < b

    def lessThanEqual(self, a, b):
        '''
        determines whether a <= b
        '''
        self.numComparisons += 1
        return a <= b

    def greaterThan(self, a, b):
        '''
        determines whether a > b
        '''
        self.numComparisons += 1
        return a > b

    def greaterThanEqual(self, a, b):
        '''
        determines whether a >= b
        '''
        self.numComparisons += 1
        return a >= b

    def equal(self, a, b):
        '''
        determines whether a == b
        '''
        self.numComparisons += 1
        return a == b

    def swap(self, index1, index2):
        '''
        swaps the data[index1] and data[index2]
        '''
        self.numSwaps += 1
        self.data[index1], self.data[index2] = self.data[index2], self.data[
            index1]
```

A.2 Classic Quicksort

```
from BaseQuicksort import BaseQuicksort

__author__ = 'paymahnmoghadasian'

class ClassicQuicksort(BaseQuicksort):
    def __init__(self, data, doInsertionSort=False, insertionSortThreshold=10,
        pivotSelection=1):
        '''
        :param pivotSelection: Determines how the pivot should be chosen. 1 = 1
            st in range. 2 = last in range. 3 = median of first, middle and last
             in range.
        '''
        if pivotSelection != 1 and pivotSelection != 2 and pivotSelection != 3:
            raise ValueError("The value of the pivot selection (%d) is invalid.
                Must be 1 or 2." % self.pivotSelection)


        BaseQuicksort.__init__(self, data, doInsertionSort,
            insertionSortThreshold, pivotSelection, 1)


    def sort(self):
        self.__sort(0, len(self.data))
        return self.data

    def __sort(self, lower, upper):
        #check whether we should do an insertion sort instead of quicksort
        if self.doInsertionSort and upper - lower <= self.insertionSortThreshold
            :
            self._insertionSort(self.data, lower, upper)
            return

        #base cases
        if upper - lower <= 1:
            return
        if upper - lower == 2:
            if self.lessThan(self.data[upper-1], self.data[lower]):
                self.swap(lower, upper-1)
            return
```

```
        # recursive case
        pivot = self.__selectPivot(lower, upper)
        i = lower+1
        for j in xrange(lower+1, upper):
            if self.lessThan(self.data[j], pivot):
                self.swap(i, j)
                i += 1

        #do the recursive calls
        self.swap(lower, i-1)
        self.__sort(lower, i-1)
        self.__sort(i, upper)

    def __selectPivot(self, lower, upper):
        if self.pivotSelection == 2:
            self.swap(lower, upper-1)
        elif self.pivotSelection == 3:
            middle = lower + (upper - lower) / 2
            pivots = self._insertionSort([(self.data[lower], lower), (self.data[
                middle], middle), (self.data[upper-1], upper-1)], 0, 3)
            self.swap(lower, pivots[1][1])

        return self.data[lower]
```

A.3 Dual Pivot Quicksort

```
__author__ = 'paymahnmoghadasian'

from BaseQuicksort import BaseQuicksort


class DualPivotQuicksort(BaseQuicksort):

    def __init__(self, data, doInsertionSort=False, insertionSortThreshold=10,
        pivotSelection=1, behaveOptimally=False):
        '''
        :param pivotSelection: This determines how pivots are chosen. There are
            exactly 2 valid values: 1 and 2. 1 is the default where the first
            and last values in the range are chosen. 2 is where tertiles are
            chosen.
        '''
```

```
        if pivotSelection != 1 and pivotSelection != 2:
            raise ValueError("The value of the pivot selection (%d) is invalid.
                Must be 1 or 2." % pivotSelection)

        BaseQuicksort.__init__(self, data, doInsertionSort,
            insertionSortThreshold, pivotSelection, 2)

        self.__behaveOptimally = behaveOptimally

    def sort(self):
        self.__sort(0, len(self.data))

    def __partitionOptimally(self, largePivot, lower, smallPivot, upper):
        lowerSwap = i = lower + 1
        upperSwap = upper - 2
        smallCount = largeCount = 0 # number of elements smaller than the small
            pivot and larger than the large pivot

        while i <= upperSwap:
            if smallCount >= largeCount:
                if self.lessThan(self.data[i], smallPivot):
                    self.swap(i, lowerSwap)
                    lowerSwap += 1
                    smallCount += 1
                elif self.greaterThan(self.data[i], largePivot):
                    #don't want to swap stuff that's bigger than the largePivot
                    while i < upperSwap and self.greaterThan(self.data[upperSwap
                        ], largePivot):
                        upperSwap -= 1

                    self.swap(i, upperSwap)
                    upperSwap -= 1
                    largeCount += 1

                    if self.lessThan(self.data[i], smallPivot):
                        self.swap(i, lowerSwap)
                        lowerSwap += 1
                        smallCount += 1
            else:
                if self.greaterThan(self.data[i], largePivot):
                    #don't want to swap stuff that's bigger than the largePivot
```

```
                while i < upperSwap and self.greaterThan(self.data[upperSwap
                    ], largePivot):
                    upperSwap -= 1

                self.swap(i, upperSwap)
                upperSwap -= 1
                largeCount += 1

                if self.lessThan(self.data[i], smallPivot):
                    self.swap(i, lowerSwap)
                    lowerSwap += 1
                    smallCount += 1

            elif self.lessThan(self.data[i], smallPivot):
                self.swap(i, lowerSwap)
                lowerSwap += 1
                smallCount += 1

        i += 1

    return lowerSwap, upperSwap


def __partition(self, largePivot, lower, smallPivot, upper):

    lowerSwap = i = lower + 1
    upperSwap = upper - 2

    if self.__behaveOptimally:
        lowerSwap, upperSwap = self.__partitionOptimally(largePivot, lower,
            smallPivot, upper)
    else:
        while i <= upperSwap:
            if self.lessThan(self.data[i], smallPivot):
                self.swap(i, lowerSwap)
                lowerSwap += 1
            elif self.greaterThan(self.data[i], largePivot):
                #don't want to swap stuff that's bigger than the largePivot
                while i < upperSwap and self.greaterThan(self.data[upperSwap
                    ], largePivot):
                    upperSwap -= 1
                self.swap(i, upperSwap)
```

```
                upperSwap -= 1

                if self.lessThan(self.data[i], smallPivot):
                    self.swap(i, lowerSwap)
                    lowerSwap += 1

            i += 1

        return lowerSwap, upperSwap

    def __sort(self, lower, upper):

        #check whether we should do an insertion sort instead of quicksort
        if self.doInsertionSort and upper - lower <= self.insertionSortThreshold
            :
            self._insertionSort(self.data, lower, upper)
            return

        #base cases
        if upper - lower <= 1:
            return
        if upper - lower == 2:
            if self.lessThan(self.data[upper-1], self.data[lower]):
                self.swap(lower, upper-1)
            return

        # select the pivots. If they're the same, this section is sorted
        smallPivot, largePivot = self.__selectPivots(lower, upper)
        # if self.equal(smallPivot, largePivot):
        #     return

        lowerSwap, upperSwap = self.__partition(largePivot, lower, smallPivot,
            upper)

        self.swap(lower, lowerSwap - 1)
        self.swap(upper - 1, upperSwap + 1)

        self.__sort(lower, lowerSwap-1)
        self.__sort(lowerSwap, upperSwap+1)
        self.__sort(upperSwap+2, upper)

    def __selectPivots(self, lower, upper):
```

```
        '''
        Default pivot selection guarantees that if the pivots can be different,
            they will be
        Tertiles does NOT make this guarantee
        '''

        # to do tertiles pivot selection, the range must be big enough
        if upper - lower >= 5 and self.pivotSelection == 2:
            middle = lower + (upper - lower) / 2
            left = lower + (middle - lower) / 2
            right = middle + (upper - middle) / 2

            #sort the potential pivot values and keep track of their index
            pivots = [(self.data[lower], lower), (self.data[left], left), (self.
                data[middle], middle),
                      (self.data[right], right), (self.data[upper - 1], upper -
                          1)]
            pivots = self._insertionSort(pivots, 0, len(pivots))

            #put the desired pivots at the beginning and end of the range
            self.swap(lower, pivots[1][1])
            if self.equal(lower, pivots[3][1]):
                self.swap(upper - 1, pivots[1][1])
            else:
                self.swap(upper - 1, pivots[3][1])

        else:
            i = lower
            while self.equal(self.data[i], self.data[upper-1]) and i < upper:
                i += 1

            self.swap(i, lower)

            if self.lessThan(self.data[upper - 1], self.data[lower]):
                self.swap(lower, upper - 1)

        return self.data[lower], self.data[upper - 1]
```

A.4 Three Pivot Quicksort

```
__author__ = 'paymahn'
```

```
from BaseQuicksort import BaseQuicksort


class ThreePivotQuicksort(BaseQuicksort):

    def __init__(self, data, insertionSortThreshold=13):
        BaseQuicksort.__init__(self, data, True, insertionSortThreshold, 1, 3)

    def sort(self):
        self.__sort(0, len(self.data)-1)

    def __sort(self, left, right):

        if self.doInsertionSort and right - left < self.insertionSortThreshold:
            self.data = self._insertionSort(self.data, left, right+1)
            return

        smallPivot, middlePivot, largePivot  = self.__selectPivots(left, right)

        a,b,c,d = self.__partition(smallPivot, middlePivot, largePivot, left,
            right)

        self.__sort(left, a-1)
        self.__sort(a+1, b-1)
        self.__sort(b+1, d-1)
        self.__sort(d+1,right)

    def __partition(self, smallPivot, middlePivot, largePivot, left, right):
        a = b = left + 2
        c = d = right - 1

        while self.lessThanEqual(b,c):
            while self.lessThan(self.data[b], middlePivot) and self.
                lessThanEqual(b, c):
                if self.lessThan(self.data[b], smallPivot):
                    self.swap(a,b)
                    a += 1
                b += 1
            while self.greaterThan(self.data[c], middlePivot) and self.
                lessThanEqual(b,c):
                if self.greaterThan(self.data[c], largePivot):
```

```
                    self.swap(c,d)
                    d -= 1
                c -= 1
            if self.lessThanEqual(b, c):
                if self.greaterThan(self.data[b], largePivot):
                    if self.lessThan(self.data[c], smallPivot):
                        self.swap(b,a)
                        self.swap(a,c)
                        a += 1
                    else:
                        self.swap(b,c)
                    self.swap(c,d)
                    b += 1
                    c -= 1
                    d -= 1
                else:
                    if self.lessThan(self.data[c], smallPivot):
                        self.swap(b,a)
                        self.swap(a,c)
                        a += 1
                    else:
                        self.swap(b,c)
                    b += 1
                    c -= 1

        a -= 1
        b -= 1
        c += 1
        d += 1
        self.swap(left + 1, a)
        self.swap(a,b)
        a -= 1
        self.swap(left, a)
        self.swap(right,d)

        return a,b,c,d

    def __movePivots(self, pivots, lower, upper):
        '''
        both bounds are inclusive
        moves pivots[0] to lower, pivots[1] to lower + 1 and pivots[2] to upper
```

```
        '''
        pivotIndices = [pivot[1] for pivot in pivots]

        currPivot = pivotIndices[0]
        self.swap(lower, currPivot)

        # if one the remaining pivots was previously located at lower, it's now
            located at currPivot
        pivotIndices = [currPivot if index == lower else index for index in
            pivotIndices[1:]]

        currPivot = pivotIndices[0]
        self.swap(lower + 1, currPivot)

        # if one the remaining pivots was previously located at lower+1, it's
            now located at currPivot
        pivotIndices = [currPivot if index == lower+1 else index for index in
            pivotIndices[1:]]

        currPivot = pivotIndices[0]
        self.swap(currPivot, upper)




    def __selectPivots(self, lower, upper):
        '''
        returns the indices of the 3 pivots we want to choose
        '''
        diff = upper - lower
        jump = diff / 6
        if jump < 1:
            raise ValueError("Cannot select pivots on a range less than 7 values
                wide")

        if diff % 6 != 0:
            upper -= diff % 6

        potentialPivots = []

        for i in range(lower, upper+1, jump):
```

```
            potentialPivots.append((self.data[i], i))

        if len(potentialPivots) != 7:
            raise RuntimeError("Shit broke. We expected 7 potential pivots")

        potentialPivots = self._insertionSort(potentialPivots, 0, 7)

        pivots = [potentialPivots[1], potentialPivots[3], potentialPivots[5]]
        self.__movePivots(pivots, lower, upper + diff % 6) # + diff % 6 because
            we subtract that about 10 lines higher

        return [pivot[0] for pivot in pivots] #return the actual pivot values
```

A.5 Yaroslavskiy Quicksort

```
__author__ = 'paymahnmoghadasian'

# http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf

from BaseQuicksort import BaseQuicksort

class YaroslavskiyQuicksort(BaseQuicksort):

    INSERTION_SORT_THRESHOLD = 17
    DIST_SIZE = 13

    def __init__(self, data):
        BaseQuicksort.__init__(self, data, True, YaroslavskiyQuicksort.
            INSERTION_SORT_THRESHOLD, 1, 2)

    def sort(self):
        self.__sort(0, len(self.data) - 1)

    def __sort(self, left, right):
        '''
        Note that the upper bound here is INCLUSIVE
        unlike other sort implementations
        '''
        len = right - left

        if len < YaroslavskiyQuicksort.INSERTION_SORT_THRESHOLD:
```

```
        self._insertionSort(self.data, left, right + 1) #plust 1 because
            insertion sort has an exclusive upper bound
        return

    sixth = len / 6
    m1 = left + sixth
    m2 = m1 + sixth
    m3 = m2 + sixth
    m4 = m3 + sixth
    m5 = m4 + sixth

    if self.greaterThan(self.data[m1], self.data[m2]):
        self.swap(m1,m2)
    if self.greaterThan(self.data[m4], self.data[m5]):
        self.swap(m4,m5)
    if self.greaterThan(self.data[m1], self.data[m3]):
        self.swap(m1,m3)
    if self.greaterThan(self.data[m2], self.data[m3]):
        self.swap(m2,m3)
    if self.greaterThan(self.data[m1], self.data[m4]):
        self.swap(m1,m4)
    if self.greaterThan(self.data[m3], self.data[m4]):
        self.swap(m3,m4)
    if self.greaterThan(self.data[m2], self.data[m5]):
        self.swap(m2,m5)
    if self.greaterThan(self.data[m2], self.data[m3]):
        self.swap(m2,m3)
    if self.greaterThan(self.data[m4], self.data[m5]):
        self.swap(m4,m5)


    self.swap(m2, left)
    self.swap(m4, right)
    pivot1 = self.data[left]
    pivot2 = self.data[right]

    diffPivots = pivot1 != pivot2

    less = left + 1
    great = right — 1

    if diffPivots:
```

```
            k = less
            while k <= great:
                x = self.data[k]

                if self.lessThan(x, pivot1):
                    self.swap(k, less)
                    less += 1
                elif self.greaterThan(x, pivot2):
                    while self.greaterThan(self.data[great], pivot2) and k <
                        great:
                        great -= 1
                    self.swap(k, great)
                    great -= 1
                    x = self.data[k]

                    if self.lessThan(x, pivot1):
                        self.swap(k, less)
                        less += 1

                k += 1

        else:
            k = less
            while k <= great:

                x = self.data[k]

                if self.equal(x, pivot1):
                    continue
                if self.lessThan(x, pivot1):
                    self.swap(k, less)
                    less += 1
                elif self.greaterThan(x, pivot2):
                    while self.greaterThan(self.data[great], pivot2) and k <
                        great:
                        great -= 1
                    self.swap(k, great)
                    great -= 1
                    x = self.data[k]

                    if self.lessThan(x, pivot1):
                        self.swap(k, less)
```

```
                    less += 1

            k += 1

        # swap
        self.swap(left, less -1)
        self.swap(right, great + 1)

        #recursive calls
        self.__sort(left, less - 2)
        self.__sort(great + 2, right)

        if great - less > len - YaroslavskiyQuicksort.DIST_SIZE and diffPivots:
            k = less
            while k <= great:

                x = self.data[k]

                if self.equal(x, pivot1):
                    self.swap(k, less)
                    less += 1
                elif self.equal(x, pivot2):
                    self.swap(k, great)
                    great -= 1
                    x = self.data[k]

                    if self.equal(x, pivot1):
                        self.swap(k, less)
                        less += 1

                k += 1
        if diffPivots:
            self.__sort(less, great)
```

A.6 M Pivot Quicksort

```
__author__ = 'paymahn'

from BaseQuicksort import BaseQuicksort
```

```
class MPivotQuicksort(BaseQuicksort):

    INSERTION_SORT_THRESHOLD = 13

    def __init__(self, data, numPivots, minHeapOptimization=False):
        if numPivots <= 0 or (data is not None and 2*numPivots > len(data) and
            2*numPivots > MPivotQuicksort.INSERTION_SORT_THRESHOLD):
            raise ValueError("Invalid value for the number of pivots. Must be
                greater than 0 and less than half the length of the data to be
                sorted")

        BaseQuicksort.__init__(self, data, True, MPivotQuicksort.
            INSERTION_SORT_THRESHOLD, 1, numPivots)
        self.__minHeapOptimization = minHeapOptimization

    def sort(self):
        self.__sort(0, len(self.data) - 1)

    def __minHeapify(self, first, last):
        '''
        make self.data[first] to self.data[last] satisfy the min heap property
        :param first: inclusive lower bound on what part of the data should be
            heapified
        :param last: inclusive upper bound on what part of the data should be
            heapified
        '''
        diff = last - first + 1
        offset = first
        for i in range(diff):
            leftChildIndex = 2 * i + 1
            rightChildIndex = 2 * i + 2

            hasLeftChild = leftChildIndex < diff
            hasRightChild = rightChildIndex < diff

            if not hasLeftChild:
                # curernt element has no "children". No further elements will
                    either
                break

            minChildIndex = leftChildIndex
```

```
            if hasRightChild and self.lessThan(self.data[rightChildIndex +
                offset], self.data[leftChildIndex + offset]):
                minChildIndex = rightChildIndex

            if self.greaterThan(self.data[i + offset], self.data[minChildIndex +
                 offset]):
                self.swap(i + offset, minChildIndex + offset)



    def __sort(self, first, last):
        if first >= last or first < 0:
            return

        if last - first < MPivotQuicksort.INSERTION_SORT_THRESHOLD:
            self._insertionSort(self.data, first, last + 1)
            return

        if self.__minHeapOptimization:
            self.__minHeapify(first, last)

        pivots = self.__choosePivots(first, last)


        pivots = self._insertionSort(pivots, 0, len(pivots))
        self._insertionSort(self.data, pivots[0]-1, last + 1)

        nextStart = first
        for i, currPivot in enumerate(pivots):
            nextGreater = nextStart
            nextGreater = self.__partition(nextStart, nextGreater, currPivot)
            self.swap(nextGreater, currPivot)
            pivots[i] = nextGreater
            self.swap(nextGreater + 1, currPivot + 1)

            if self.equal(nextStart, first) and self.greaterThan(pivots[i],
                nextStart + 1):
                self.__sort(nextStart, pivots[i] - 1)

            if not self.equal(nextStart, first) and self.greaterThan(pivots[i],
                pivots[i-1] + 2):
                self.__sort(pivots[i-1]+1, pivots[i]+1)
```

```
            nextStart = nextGreater + 2
        if self.greaterThan(last, pivots[-1] + 1):
            self.__sort(pivots[-1]+1, last)



    def __choosePivots(self, first, last):
        pivots = range(self.numPivots)

        size = last - first + 1
        segments = self.numPivots + 1
        candidate = size / segments - 1

        next = 2
        if candidate >= 2:
            next = candidate + 1

        candidate += first
        for i in range(self.numPivots):
            pivots[i] = candidate
            candidate += next

        for i in reversed(range(self.numPivots)):
            self.swap(pivots[i]+1, last)
            last -= 1
            self.swap(pivots[i], last)
            pivots[i] = last
            last -= 1


        return pivots

    def __partition(self, nextStart, nextGreater, curPivot):
        for curUnknown in range(nextStart, curPivot):
            if self.lessThan(self.data[curUnknown], self.data[curPivot]):
                self.swap(curUnknown, nextGreater)
                nextGreater += 1
        return nextGreater
```

B Data Analysis Code

```
from matplotlib import pyplot as plt
import numpy as np
from scipy.optimize import curve_fit

fileName = 'alldata.csv'

dataAbsPath = '../quicksorts/' + fileName

DEBUG = False

def getData(filePath):

    flink = open(filePath,'r')

    data = {}

    header = "Name,Length,Median Selection,Num Pivots,Used Insertion Sort,
        Insertion Sort Threshold,Time,Comparisons,Swaps\n"

    isFirst = True

    for line in flink:
        if line not in header and len(line) > 0 and line!=header:

            isNoError = True

            temp = line.strip().split(',')

            name = temp[0]

            try:
                length = int(temp[1])
            except:
                print "Error:",temp[1]
                isNoError = False

            try:
                medianSelection = int(temp[2])
            except:
                print "Error:",temp[2]
```

```
        isNoError = False

    try:
        numPivots = int(temp[3])
    except:
        print "Error:",temp[3]
        isNoError = False

    try:
        usedInsertionSort = bool(temp[4])
    except:
        print "Error:",temp[4]
        isNoError = False

    try:
        insertionSortThreshold = int(temp[5]) # will not use
    except:
        print "Error:",temp[5]
        isNoError = False

    try:
        time = float(temp[6])
    except:
        print "Error:",temp[6]
        isNoError = False

    try:
        comparisons = int(temp[7])
    except:
        print "Error:",temp[7]
        isNoError = False

    try:
        swaps = int(temp[8])
    except:
        print "Error:",temp[8]
        isNoError = False

    if isNoError:
        label = ( name,medianSelection,numPivots,usedInsertionSort)

        if label in data :
```

```
                    sizeList,timeList,compList,swapList = data[label]
                else:
                    sizeList = []
                    timeList = []
                    compList = []
                    swapList = []

                sizeList.append(length)
                timeList.append(time)
                compList.append(comparisons)
                swapList.append(swaps)

                data[label] = sizeList,timeList,compList,swapList

    for label in data.keys() :
        sizeList,timeList,compList,swapList = data[label]

        tempList = zip(sizeList,timeList,compList,swapList)

        tempList.sort()

        sizeList = [ item[0] for item in tempList]
        timeList = [ item[1] for item in tempList]
        compList = [ item[2] for item in tempList]
        swapList = [ item[3] for item in tempList]


        sizeList = np.array(sizeList, dtype = np.int64)
        timeList = np.array(timeList, dtype = np.float64)
        compList = np.array(compList, dtype = np.int64)
        swapList = np.array(swapList, dtype = np.int64)

        data[label] = sizeList,timeList,compList,swapList

    return averageData(data)

def averageData(data):
    '''
    This function will take in the data dictionary and average all the data
    points with the same size value
    '''
```

```
    for label in data.keys() :
        sizeList,timeList,compList,swapList = data[label]


        sizeListTemp = []
        timeListTemp = []
        compListTemp = []
        swapListTemp = []

        for size in np.nditer(sizeList):

            if size not in sizeListTemp:
                sizeListTemp.append(size)

                indexArray = size == sizeList
                numValues = len(indexArray) * 1.0

                # The averages
                timeTemp = np.sum(timeList[ indexArray ])/numValues
                compTemp = np.sum(compList[ indexArray ])/numValues
                swapTemp = np.sum(swapList[ indexArray ])/numValues

                timeListTemp.append(timeTemp)
                compListTemp.append(compTemp)
                swapListTemp.append(swapTemp)


        sizeList = np.array(sizeListTemp,dtype = np.int64)
        timeList = np.array(timeListTemp,dtype = np.float128)
        compList = np.array(compListTemp,dtype = np.int64)
        swapList = np.array(swapListTemp,dtype = np.int64)

        data[label] = sizeList,timeList,compList,swapList

    return data

def markerGenerator(index,withSymbol= True, withColor = True):
    '''
    This function was created so that we can generate lines
    with varying symbols and colors without creating them by
    hand.
    '''
```

```
    colors = 'rbgmcky'
    numColors = len(colors)
    markers = 'ox^spdv><'
    numMarkers = len(markers)

    return withSymbol*markers[index%numMarkers]+withColor*colors[index%numColors
        ]

def convertLabelToStr(label):
    name,medianSelection,numPivots,usedInsertionSort = label
    return "%s - %s - %s" %(name,medianSelection,numPivots)

def plotData(data,  plotTime = False,
                    plotComp = True,
                    plotSwap = True,
                    goodFunction = lambda x:True ,
                    badFunction = lambda x:False,
                    makeLegend = True,
                    legendSize = 10,
                    plotTitle = None,
                    fontsize = 12,
                    plotter = plt.plot,
                    connectDataPoints = False,
                    xlim = None,
                    specialFlag = False) :
    '''
    Plot data will take the data dictionary and create plots according to the
        key word argunments.

    Note that labels are defined as follows (name,medianSelection,numPivots,
        usedInsertionSort)

    :param data: the dictionary that contains data to be plotted
    :param plotTime: boolean to control if the size vs time plots will actually
        be rendered
    :param plotComp: boolean to control if the size vs comparisons plots will
        actually be rendered
    :param plotSwap: boolean to control if the size vs swaps plots will actually
         be rendered
    :param goodFunction: a function that will take in the label tuple and
        determine if it will plot that label
```

```
    :param badFunction:  a function that will take in the label tuple and
        determine if it will not plot that label
    :param makeLegend: boolean to control if the legend should be rendered
    :param legendSize: size of the legend
    :param plotTitle: string of title of the plot. Will be placed on the top of
        the figure.
    :param fontsize: The font size of the title of the figure
    :param plotter: function that will plot the data ( plt.plot, plt.semilogx,
        plt.semilogy, plt.loglog )
    :param connectDataPoints: boolean to control if the data points will be
        connected with a solid line
    '''

    keyList = list(data.keys())
    keyList.sort()

    #print keyList

    if plotTime :
        timeFigure = plt.figure()
    if plotComp :
        compFigure = plt.figure()
    if plotSwap :
        swapFigure = plt.figure()

    if xlim and xlim[0] > xlim[1]:

        xlim[1],xlim[0] = xlim[0],xlim[1]

    for count,label in enumerate(keyList):

        marker = connectDataPoints*'-' + markerGenerator(count)

        if goodFunction(label) and not badFunction(label) :
            sizeList,timeList,compList,swapList = data[label]

            if specialFlag :
                timeList = timeList/( sizeList*np.log2(sizeList) )
                compList = compList/( sizeList*np.log2(sizeList) )
                swapList = swapList/( sizeList*np.log2(sizeList) )
                sizeList = np.log2(sizeList)
```

```
            if plotTime :
                plt.figure(timeFigure.number)
                plotter(sizeList,timeList,marker,label=convertLabelToStr(label))
            if plotComp :
                plt.figure(compFigure.number)
                plotter(sizeList,compList,marker,label=convertLabelToStr(label))
            if plotSwap :
                plt.figure(swapFigure.number)
                plotter(sizeList,swapList,marker,label=convertLabelToStr(label))

            if DEBUG:
                index = ( xlim[0] <= sizeList) * (sizeList <= xlim[1] )
                print ""
                print convertLabelToStr(label)
                print sizeList[index]
                print compList[index]
                print swapList[index]

    returnList = []

    legendProp = {'size':legendSize}

    if xlim:
        timeYLim,compYLim,swapYLim = extractYLim(data, goodFunction, badFunction
            , xlim)

    if plotTime :
        returnList.append(timeFigure)
        plt.figure(timeFigure.number)
        plt.xlabel('Size')
        plt.ylabel('Time')

        if makeLegend :
            plt.legend(loc = "upper left",prop = legendProp)

        if plotTitle:
            plt.title(plotTitle + ' (Time)',fontsize = fontsize)

        if xlim:
            plt.xlim(xlim[0],xlim[1])
            plt.ylim(timeYLim[0],timeYLim[1])
```

```
    if plotComp :
        returnList.append(compFigure)
        plt.figure(compFigure.number)
        plt.xlabel('Size')
        plt.ylabel('Comparisons')

        if makeLegend :
            plt.legend(loc = "upper left",prop = legendProp)

        if plotTitle:
            plt.title(plotTitle + ' (Comparisons)',fontsize = fontsize)

        if xlim:
            plt.xlim(xlim[0],xlim[1])
            plt.ylim(compYLim[0],compYLim[1])

    if plotSwap:
        returnList.append(swapFigure)
        plt.figure(swapFigure.number)
        plt.xlabel('Size')
        plt.ylabel('Swaps')

        if makeLegend :
            plt.legend(loc = "upper left",prop = legendProp)

        if plotTitle:
            plt.title(plotTitle + ' (Swaps)',fontsize = fontsize)

        if xlim:
            plt.xlim(xlim[0],xlim[1])
            plt.ylim(swapYLim[0],swapYLim[1])

    if specialFlag and plotTime:
        plt.figure(timeFigure.number)
        plt.xlabel('log(Size)')
        plt.ylabel('Time / (Size log(Size) )')

    if specialFlag and plotComp:
        plt.figure(compFigure.number)
        plt.xlabel('log(Size)')
        plt.ylabel('Comparisons / (Size log(Size) )')
```

```
    if specialFlag and plotSwap:
        plt.figure(swapFigure.number)
        plt.xlabel('log(Size)')
        plt.ylabel('Swaps / (Size log(Size) )')

    return tuple(returnList)

def plotPolynomialFit(data,fitParameters,figureList,
                      plotComp = True,
                      plotSwap = True,
                      goodFunction = lambda x:True ,
                      badFunction = lambda x:False,
                      makeLegend = True,
                      legendSize = 10,
                      plotter = plt.plot,
                      xlim = None,
                      linewidth = 1.5,
                      numPoints = 10**3,
                      specialFlag = False) :

    keyList = list(data.keys())
    keyList.sort()

    #print keyList
    count = 0
    if plotComp :
        compFigure = figureList[count]
        count+=1
    if plotSwap :
        swapFigure = figureList[count]
        count+=1

    if xlim and xlim[0] > xlim[1]:

        xlim[1],xlim[0] = xlim[0],xlim[1]

    for count,label in enumerate(keyList):

        if count < 7 :
            marker = '--' + markerGenerator(count,withSymbol=False)
        else:
            marker = '-' + markerGenerator(count,withSymbol=False)
```

```
    if goodFunction(label) and not badFunction(label) :
        sizeList,timeList,compList,swapList = data[label]
        compCoef,compCov,swapCoef,swapCov = fitParameters[label]

        if xlim :
            xMin = min( min(sizeList),xlim[0])
            xMax = max( max(sizeList),xlim[1])
        else :
            xMin = min(sizeList)
            xMax = max(sizeList)

        xVals = np.linspace(xMin,xMax,numPoints)

        if plotComp :
            compFitFunc = lambda xx:fitFunction(xx,*tuple(compCoef))

            if specialFlag :
                # Plot something special
                yVals = compFitFunc(xVals) / ( xVals * np.log2(xVals) )
                xVals = np.log2(xVals)
            else:
                yVals = compFitFunc(xVals)

            plt.figure(compFigure.number)
            plotter(xVals,yVals,marker,label=convertLabelToStr(label)+" Fit"
                ,linewidth = linewidth)

        if plotSwap :
            swapFitFunc = lambda xx:fitFunction(xx,*tuple(swapCoef))

            if specialFlag :
                # Plot something special
                yVals = swapFitFunc(xVals) / ( xVals * np.log2(xVals) )
                xVals = np.log2(xVals)
                print yVals
            else:
                yVals = swapFitFunc(xVals)

            plt.figure(swapFigure.number)
            plotter(xVals,yVals,marker,label=convertLabelToStr(label)+" Fit"
                ,linewidth = linewidth)
```

```
    legendProp = {'size':legendSize}

    if xlim:
        timeYLim,compYLim,swapYLim = extractYLim(data, goodFunction, badFunction
            , xlim)

    if plotComp and makeLegend:
        plt.figure(compFigure.number)
        plt.legend(loc = "upper left",prop = legendProp)

    if plotSwap and xlim:
        plt.xlim(xlim[0],xlim[1])
        plt.ylim(compYLim[0],compYLim[1])

    if plotSwap and makeLegend:
        plt.figure(swapFigure.number)
        plt.legend(loc = "upper left",prop = legendProp)

    if plotSwap and xlim:
        plt.xlim(xlim[0],xlim[1])
        plt.ylim(swapYLim[0],swapYLim[1])

def extractYLim(data,goodFunction,badFunction,xlim):
    keyList = list(data.keys())
    keyList.sort()

    if xlim[0] > xlim[1]:

        xlim[1],xlim[0] = xlim[0],xlim[1]

    yMinTime = None
    yMaxTime = -1

    yMinComp = None
    yMaxComp = -1

    yMinSwap = None
    yMaxSwap = -1

    for label in keyList:
        if goodFunction(label) and not badFunction(label) :
```

```
            sizeList,timeList,compList,swapList = data[label]

            sizeIndex = (xlim[0] <= sizeList) * ( sizeList <= xlim[1]  )

            yMinTimeTemp = np.min(timeList[sizeIndex])
            yMaxTimeTemp = np.max(timeList[sizeIndex])

            yMinCompTemp = np.min(compList[sizeIndex])
            yMaxCompTemp = np.max(compList[sizeIndex])

            yMinSwapTemp = np.min(swapList[sizeIndex])
            yMaxSwapTemp = np.max(swapList[sizeIndex])

            if not bool(yMinTime) or yMinTime > yMinTimeTemp:
                yMinTime = yMinTimeTemp

            if yMaxTime < yMaxTimeTemp:
                yMaxTime = yMaxTimeTemp

            if not bool(yMinComp) or yMinComp > yMinCompTemp:
                yMinComp = yMinCompTemp

            if yMaxComp < yMaxCompTemp:
                yMaxComp = yMaxCompTemp

            if not bool(yMinSwap) or yMinSwap > yMinSwapTemp:
                yMinSwap = yMinSwapTemp

            if yMaxSwap < yMaxSwapTemp:
                yMaxSwap = yMaxSwapTemp


    if not bool(yMinTime):
        yMinTime = 0

    if not bool(yMinComp):
        yMinComp = 0

    if not bool(yMinSwap):
        yMinSwap = 0

    return [yMinTime,yMaxTime],[yMinComp,yMaxComp],[yMinSwap,yMaxSwap]
```

```
def calcLeastSquaresOnData(data):
    '''
    We want to fit the data to :
        y = A x log(x) + B x + C log(x)

    We use a non—linear curve fitter.

    Other considerations:

    Method 2 :
        y = A x log(x) + B

        So we make a transformation so that :
            X = x log(x) = x ln(x)/ln(2)
            Y = y

    Method 3:
        y = A x log(x) + Bx = x ( A log(x) + B )

        So we make a transformation so that :
            X = log(x) = ln(x)/ln(2)
            Y = y/x

    Method 3:
        y = A x log(x) + Blog(x) = log(x) ( A x + B )

        So we make a transformation so that :
            X = x
            Y = y/log(x)
    '''
    keyList = list(data.keys())
    keyList.sort()

    fitParameters = {}

    for label in keyList :
        sizeList,timeList,compList,swapList = data[label]

        sizeList = np.array(sizeList, dtype = np.float64)
        timeList = np.array(timeList, dtype = np.float64)
```

```
        compList = np.array(compList, dtype = np.float64)
        swapList = np.array(swapList, dtype = np.float64)

        compCoef,compCov = curve_fit(fitFunction, sizeList, compList)
        swapCoef,swapCov = curve_fit(fitFunction, sizeList, swapList)

        fitParameters[label] = compCoef,compCov,swapCoef,swapCov

    printSpecifier = "%40s | %9.5f +-%9.5f | %9.5f +-%9.5f | %10.5f +- %9.5f "

    # xxxCov = The estimated covariance of optimal values.
    #            The diagonals provide the variance of the parameter estimate.

    # http://stats.stackexchange.com/questions/50830/can-i-convert-a-covariance-
        matrix-into-uncertainties-for-variables

    print ""
    print "COMPARISON COEFFICIENTS"
    for label in keyList :
        compCoef,compCov,swapCoef,swapCov = fitParameters[label]

        compCov[compCov>=0] = np.sqrt(compCov[compCov>=0])

        print printSpecifier%(convertLabelToStr(label),compCoef[0],compCov[0,0],
            compCoef[1],compCov[1,1],compCoef[2],compCov[2,2])

        compCov[compCov>=0] *= compCov[compCov>=0]

    print ""
    print "SWAP COEFFICIENTS"
    for label in keyList :
        compCoef,compCov,swapCoef,swapCov = fitParameters[label]

        swapCov[swapCov>=0] = np.sqrt(swapCov[swapCov>=0])

        print printSpecifier%(convertLabelToStr(label),swapCoef[0],swapCov[0,0],
            swapCoef[1],swapCov[1,1],swapCoef[2],swapCov[2,2])

        swapCov[swapCov>=0] *= swapCov[swapCov>=0]

    return fitParameters
```

```
def fitFunction(xx,AA,BB,CC):
    return AA*xx*np.log2(xx)+BB*xx+CC*np.log2(xx)

def saveFigure(figure,fileName,fileExtention = '.png',dpi = 600):
    fullFileName = fileName + fileExtention
    plt.figure(figure.number)
    plt.savefig(fullFileName,
                    dpi         = dpi,
                    facecolor   = 'w',
                    edgecolor   = 'w',
                    orientation = 'portrait',
                    papertype   = None,
                    format      = None,
                    transparent = True,
                    bbox_inches = None,
                    pad_inches  = 0.15,
                    frameon     = None)

def plotDataAndFit(data,fitParameters,
                    plotComp = True,
                    plotSwap = True,
                    goodFunction = lambda x:True ,
                    badFunction = lambda x:False,
                    makeLegend = True,
                    legendSize = 10,
                    plotTitle = None,
                    fontsize = 12,
                    plotter = plt.plot,
                    xlim = None,
                    connectDataPoints = False,
                    linewidth = 1.5,
                    numPoints = 10**3,
                    savePlot = False,
                    dpi = 600,
                    specialFlag = False) :

    figureList = plotData(data,
                    plotComp = plotComp,
                    plotSwap = plotSwap,
                    goodFunction = goodFunction ,
                    badFunction = badFunction,
                    makeLegend = makeLegend,
```

```
                    legendSize = legendSize,
                    plotTitle = plotTitle,
                    xlim = xlim,
                    fontsize = fontsize,
                    plotter = plotter,
                    connectDataPoints = connectDataPoints,
                    specialFlag = specialFlag)

    if not connectDataPoints :
        plotPolynomialFit(data,fitParameters,figureList,
                    plotComp = plotComp,
                    plotSwap = plotSwap,
                    goodFunction = goodFunction ,
                    badFunction = badFunction,
                    makeLegend = makeLegend,
                    legendSize = legendSize,
                    plotter = plotter,
                    xlim = xlim,
                    linewidth = linewidth,
                    numPoints = numPoints,
                    specialFlag = specialFlag)

    if savePlot :
        fileName = "".join(plotTitle.split() )
        compFigure,swapFigure = figureList
        saveFigure(compFigure,fileName+"_comp",fileExtention = '.png',dpi = dpi)
        saveFigure(swapFigure,fileName+"_swap",fileExtention = '.png',dpi = dpi)

    return figureList

def main():
    # Note that labels are defined as follows
    # (name,medianSelection,numPivots,usedInsertionSort)
    #
    # List of all labels as of April 4
    #
    # ('ClassicQuicksort', 1, 1, True)
    # ('ClassicQuicksort', 2, 1, True)
    # ('ClassicQuicksort', 3, 1, True)
    # ('DualPivotQuicksort', 1, 2, True)
    # ('DualPivotQuicksort', 2, 2, True)
    # ('HeapOptimizedMPivotQuicksort', 1, 3, True)
```

```
# ('HeapOptimizedMPivotQuicksort', 1, 4, True)
# ('HeapOptimizedMPivotQuicksort', 1, 5, True)
# ('HeapOptimizedMPivotQuicksort', 1, 6, True)
# ('MPivotQuicksort', 1, 3, True)
# ('MPivotQuicksort', 1, 4, True)
# ('MPivotQuicksort', 1, 5, True)
# ('MPivotQuicksort', 1, 6, True)
# ('OptimalDualPivotQuicksort', 1, 2, True)
# ('OptimalDualPivotQuicksort', 2, 2, True)
# ('ThreePivotQuicksort', 1, 3, True)
# ('YaroslavskiyQuicksort', 1, 2, True)

classicQuickSortOnly             = lambda x: x[0] == 'ClassicQuicksort'
dualPivotQuicksortOnly           = lambda x: x[0] == 'DualPivotQuicksort'
heapOptimizedMPivotQuicksortOnly = lambda x: x[0] == '
    HeapOptimizedMPivotQuicksort'
mPivotQuicksortOnly              = lambda x: x[0] == 'MPivotQuicksort'
optimalDualPivotQuicksortOnly    = lambda x: x[0] == '
    OptimalDualPivotQuicksort'
threePivotQuicksortOnly          = lambda x: x[0] == 'ThreePivotQuicksort'
yaroslavskiyQuicksortOnly        = lambda x: x[0] == 'YaroslavskiyQuicksort'

onePivot = lambda x: x[2] == 1
twoPivot = lambda x: x[2] == 2
threePivot = lambda x: x[2] == 3

usedInsertionSort = lambda x: x[3]


mPivotQuicksortOnly3 = lambda x : mPivotQuicksortOnly(x) and threePivot(x)

customPlot = lambda x: classicQuickSortOnly(x) or dualPivotQuicksortOnly(x)
    or threePivotQuicksortOnly(x) or mPivotQuicksortOnly3(x)

allMPivotQuicksortKinds = lambda x: mPivotQuicksortOnly(x) or
    heapOptimizedMPivotQuicksortOnly(x)

data = getData(dataAbsPath)

fitParameters = calcLeastSquaresOnData(data)
```

```
maskFunctionList = [ classicQuickSortOnly,dualPivotQuicksortOnly,
    heapOptimizedMPivotQuicksortOnly,
                    mPivotQuicksortOnly, optimalDualPivotQuicksortOnly,
                        threePivotQuicksortOnly,
                    yaroslavskiyQuicksortOnly, onePivot,twoPivot,threePivot,
                        allMPivotQuicksortKinds]

maskFunctionTitleList = ['Classic QuickSorts','Dual Pivot Quicksorts','Heap
    Optimized M-Pivot Quicksorts',
                        'Non Optimized M-Pivot Quicksorts','Optimal Dual
                            Pivot Quicksorts','Three Pivot Quicksorts',
                        'Yaroslavskiy Quicksorts','One Pivots','Two Pivots',
                            'Three Pivots','M-Pivot Quicksorts']

smallScaleLimits = [100,1000]

plotDataAndFit(data,fitParameters, plotTitle = 'Legend Plot',
    connectDataPoints = True,legendSize=15,savePlot=True)
plotDataAndFit(data,fitParameters, plotTitle = 'All the Plots Small Scale',
    xlim = smallScaleLimits, connectDataPoints = True,makeLegend=False,
    savePlot=True)
plotDataAndFit(data,fitParameters, plotTitle = 'All the Plots Large Scale',
    connectDataPoints = True,makeLegend=False,savePlot = True)
plotDataAndFit(data,fitParameters, plotTitle = 'Semilogx All Plots Large
    Scale ', connectDataPoints = True,makeLegend=False,savePlot = True,
    plotter = plt.semilogx)

for maskFunc,plotTitle in zip(maskFunctionList,maskFunctionTitleList):
    plotDataAndFit(data, fitParameters,goodFunction = maskFunc, plotTitle =
        plotTitle+" Large Scale",savePlot = True)

for maskFunc,plotTitle in zip(maskFunctionList,maskFunctionTitleList):
    plotDataAndFit(data, fitParameters,goodFunction = maskFunc, plotTitle =
        plotTitle+" Small Scale", xlim =smallScaleLimits,connectDataPoints =
         True,savePlot = True)

plotDataAndFit(data,fitParameters, goodFunction = allMPivotQuicksortKinds,
    plotTitle = "M-Pivot Quicksorts Large Scale",savePlot = True,legendSize
    =7)
plotDataAndFit(data,fitParameters, goodFunction = allMPivotQuicksortKinds,
    plotTitle = "M-Pivot Quicksorts Small Scale",xlim = smallScaleLimits,
    connectDataPoints = True,savePlot = True,legendSize=7)
```

```
    # The special plot
    plotDataAndFit(data,fitParameters, plotTitle = 'All Plots Large Scale logn
        vs y_OVER_nlogn', connectDataPoints = True,makeLegend=False,savePlot =
        True,specialFlag = True)

    #plotDataAndFit(data,fitParameters, goodFunction = customPlot, plotTitle =
        plotTitle+" Large Scale",savePlot = True)
    #plotDataAndFit(data,fitParameters, goodFunction = customPlot, plotTitle = '
        customPlot',xlim = smallScaleLimits,connectDataPoints = True,savePlot =
        True)

    #plt.show()

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