Artem Los, CDATE 1, 2015.02.21.

Investigating time complexities of various implementations of Quick Sort algorithm

# Introduction

In this report four different implementations of the Quick Sort algorithm are going to be examined. The implementations can be characterized as follows: fixed pivot element, random pivot element, use of insertion sort algorithm when the array has reached a certain size and pure quick sort (i.e. without insertion sort adjustment). Our aim is to find time complexities experimentally.

# Method

## Background

In many situations, there may be large differences in execution time, some of which occur in the beginning (because of just-in-time) and some that occur because the experiments are spread out over a certain period of time. The approaches to handle these issues are described below.

## Java Just-in-time time difference

To handle this, I am going to calculate a mean between the time it takes for JVM to produce machine code and the time it then takes to execute that code. This is a rough average, as it depends on the number of times the sorting algorithm is going to be called. As the number increases, it might be considered to lower the average. It can be discarded also, provided that the average case is more important. This can be achieved by performing a sort operation in advance that does not contribute to the result.

## Time differences between executions

Ideally, I would test all methods in the same environment to be able to establish a more or less valid ratio. This is not feasible, so I will focus on executing methods several times, at different instances, and figure out an average.

## Method used to collect data

### Finding the fastest algorithm

In order to find the fastest algorithm, 100 arrays will be sorted with 1000 elements ranging up to the maximum value of an integer. Each execution of the test method will produce hundred data points (in nanoseconds). This is going to be repeated five times. An adapted version of the test for the QuickSort implementation with fixed pivot and no usage of insertion sort are located in Appendix A.

### Comparing the speed of Insertion Sort and Arrays.Sort

In order to compare Insertion Sort vs. Arrays.Sort (which is based on Quick Sort), speed in nanoseconds is going to be calculated for arrays of sizes  . Each execution will be performed only once. We are going to look at both random arrays and ordered arrays (in ascending and descending order).

### Finding the optimal K value (the breaking point)

By *K value* we mean the breaking point when Quick sort switches to Insertion sort. In order to find this value, Quick sort is going to be tested for *k* values from 1 to 100 by recording the time in an array (it is the same Array set up as in *Finding the fastest algorithm*). This procedure is going to be repeated 100 times, and if the execution time for a given *k* value is greater than the one in the array, this value is going to be inserted into the array instead. Thus, the array will contain the longest time that was encountered for a given *k* value during hundred executions.

Later on, the execution times for the *k* values recorded in the array is going to be inserted to MS Excel, with the aim to find the smallest time and thus the optimal *k* value. Again, this is to be repeated five times so that an average can be obtained.

# Results

## Finding fastest algorithm

Figure : Illustration of the comparison of the average execution time (ns) and the algorithm used. In this case, the algorithm refers to the different implementations of Quick Sort.

## The optimal K value

Figure 2: The different values for K - the breaking point when Insertion sort is used. The arithmetic average of the data is 15.

## Comparison between optimized version and Figure 1

Figure : The comparison between the optimized version (Fixed pivot with K=15) and the results obtained in Figure 1. NB: The optimized results and the ones in Figure 1 were obtained in different settings.

## Comparison between Fixed Pivot (insertion sort) and optimized version

Because of the fact that the results in *Figure 1* and the ones obtained for the optimized version of the algorithm were collected at different instances in time (combined into *Figure 2*), the algorithm that was closest in time to the optimized version was recalculated and juxtaposed with the fixed pivot (insertion sort) algorithm.

Figure : Optimized results and Fixed Pivot (Insertion sort) in the same environment.

## Comparison of different sizes of arrays and algorithm used

Figure : An illustration of the percentage of time used by Arrays.Sort and Insertion sort for a given array size. Note, the higher the percentage, the more time it took to sort the list.

Figure : An illustration of the percentage of time used by Arrays.Sort and Insertion sort for a given array size. Note, the higher the percentage, the more time it took to sort the list.

Figure : An illustration of the percentage of time used by Arrays.Sort and Insertion sort for a given array size. Note, the higher the percentage, the more time it took to sort the list.

# Discussion and Conclusion

## Fastest algorithm

Based on *Figure 1* and *Figure 4,* it can be concluded that the fastest implementation of Quick sort is when Insertion sort is used for a K value equal to 15. The K value is obtained by calculating the arithmetic mean of values in *Figure 2*. This may appear to be strange as the time complexity for Insertion sort is  while it is  for Quick sort. However, if we think about the hidden premise in the expression of the time complexity using Ordo notation, the time complexity actually says:



Clearly, we are not near infinity when  is small enough. Moreover, there is another hidden premise, namely that Ordo notation represents a set of all functions such that there exists a constant  :



The constant is not specified as it vanishes when  gets big enough, but when  is small, the constants may have a more significant importance.

## Comparison between Insertion Sort and Arrays.Sort

Since Arrays.Sort implements Quick Sort, it can be compared with Insertion Sort. From *Figure 5* and *Figure 7*, it can be seen that in the range of from 1 to 16, they both behave have a similar execution time. However, as the array gets larger, the Arrays.Sort method is much faster. From *Figure 6*, it can be concluded that no significant different exists between the two algorithms when the data is already ordered in ascending order.

# Appendix A – Source code

## Tests

|  |
| --- |
| **package** inda6;  **import** **static** org.junit.Assert.\*;  **import** inda6.Data.Order;  **import** java.beans.Expression;  **import** java.util.ArrayList;  **import** java.util.Random;  **import** java.util.Arrays;  **import** org.junit.Before;  **import** org.junit.Test;  **public** **class** SortTest {  /\*\*  \* t: test case, s: expected solution.  \*/  **private** **int**[] t0, s0, t1, s1, t2, s2, t7, s7;  /\*\*  \* Big array of random numbers.  \* tr: test case, sr: expected solution.  \* R\_SIZE is the size of the array.  \*/  **private** **static** **final** **int** ***R\_SIZE*** = 10000;  **private** **int**[] tr, sr;  **private** Random rand;    ISort sort;  /\*\*  \* Constructs a new test case.  \*/  @Before  **public** **void** SetUp()  {  rand = **new** Random();    t0 = **new** **int**[0];  s0 = **new** **int**[0];  t1 = **new** **int**[] {1};  s1 = **new** **int**[] {1};  t2 = **new** **int**[] {2, 1};  s2 = **new** **int**[] {1, 2};  t7 = **new** **int**[] {9, 5, 2, 7, 1, 6, 6};  s7 = **new** **int**[] {1, 2, 5, 6, 6, 7, 9};  tr = **new** **int**[***R\_SIZE***];  sr = **new** **int**[***R\_SIZE***];  **for** (**int** i = 0; i < ***R\_SIZE***; i++) {  tr[i] = sr[i] = rand.nextInt();  }  Arrays.*sort*(sr);  }    @Test  **public** **void** testAllSortAlg()  {  sort = **new** InsertionSort();  workingSortAlg();      SetUp();  sort = **new** QuickSortFixedPivot();    workingSortAlg();      SetUp();  sort = **new** QuickSortFixedPivot();    sort.setAlg(**new** InsertionSort());  workingSortAlg();    SetUp();  sort = **new** QuickSortRandPivot();    sort.setAlg(**new** QuickSortRandPivot());  workingSortAlg();  }    @Test  **public** **void** InsertionSortNewTests()  {  InsertionSort sortA = **new** InsertionSort();    **int**[] rand = **new** **int**[] {9, 5, 2, 7, 1, 6, 6};    sortA.specsort(rand, rand.length-4, rand.length);    **int**[] actual = **new** **int**[]{9, 5, 2, 1 , 6, 6, 7};  **for** (**int** i = 0; i < rand.length; i++) {  *assertTrue*(actual[i] == rand[i]);    }  }    **public** **void** workingSortAlg()  {  sort.sort(t0);  *assertTrue*(Arrays.*equals*(t0, s0));  sort.sort(t1);  *assertTrue*(Arrays.*equals*(t1, s1));  sort.sort(t2);  *assertTrue*(Arrays.*equals*(t2, s2));  sort.sort(t7);  *assertTrue*(Arrays.*equals*(t7, s7));  sort.sort(tr);  *assertTrue*(Arrays.*equals*(tr, sr));  }    @Test  **public** **void** RequiredTests()  {  //these are required in the exercise    //ALGORITHM 1  sort = **new** QuickSortFixedPivot();  sort.setAlg(**new** QuickSortFixedPivot());  RequiredTestCases();    //ALGORITHM 2  sort = **new** QuickSortFixedPivot();  sort.setAlg(**new** InsertionSort());  RequiredTestCases();    //ALGORITHM 3  sort = **new** QuickSortRandPivot();  sort.setAlg(**new** InsertionSort());  RequiredTestCases();    //ALGORITHM 3  sort = **new** QuickSortRandPivot();  sort.setAlg(**new** QuickSortRandPivot());  RequiredTestCases();    }      **public** **void** RequiredTestCases()  {  //assertTrue(false);  Data dt = **new** Data(10000, 50, Order.***RANDOM***);    **int**[] expdata = dt.get();  **int**[] actualdata = dt.get();  Arrays.*sort*(actualdata);  sort.sort(expdata);  *assertTrue*(compareArrays(expdata,actualdata));    dt = **new** Data(1000, 50, Order.***DESCENDING***);    expdata = dt.get();  actualdata = dt.get();  Arrays.*sort*(actualdata);  sort.sort(expdata);  *assertTrue*(compareArrays(expdata,actualdata));    dt = **new** Data(1000, 1, Order.***DESCENDING***);    expdata = dt.get();  actualdata = dt.get();  Arrays.*sort*(actualdata);  sort.sort(expdata);  *assertTrue*(compareArrays(expdata,actualdata));  }    **public** **boolean** compareArrays(**int**[] a, **int**[] b)  {  **if**(a.length != b.length)  **return** **false**;    **for** (**int** i = 0; i < b.length; i++) {  **if**(a[i] != b[i])  **return** **false**;  }    **return** **true**;  }    @Test  **public** **void** ExecutionTimePivotNonInsertionSort()  {    **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  sort = **new** QuickSortFixedPivot();  sort.setAlg(**new** QuickSortFixedPivot());    Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    sort.sort(expdata);  sw.stop();    System.***out***.println(sw.nanoseconds());  }  }  @Test  **public** **void** ExecutionTimePivotInsertionSort()  {    **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  sort = **new** QuickSortFixedPivot();  sort.setAlg(**new** InsertionSort());    Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    sort.sort(expdata);  sw.stop();    System.***out***.println(sw.nanoseconds());  }  }    @Test  **public** **void** ExecutionTimeRandPivotInsertionSort()  {    **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  sort = **new** QuickSortRandPivot();  sort.setAlg(**new** InsertionSort());    Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    sort.sort(expdata);  sw.stop();    System.***out***.println(sw.nanoseconds());  }  }    @Test  **public** **void** ExecutionTimeRandPivotNonInsertionSort()  {    **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  sort = **new** QuickSortRandPivot();  sort.setAlg(**new** QuickSortRandPivot());    Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    sort.sort(expdata);  sw.stop();    System.***out***.println(sw.nanoseconds());  }  }    @Test  **public** **void** OptimizedAlgorithm()  {  QuickSortFixedPivot sortA = **new** QuickSortFixedPivot();    **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  sortA = **new** QuickSortFixedPivot();  sortA.setAlg(**new** InsertionSort());  sortA.K = 15;    Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    sortA.sort(expdata);  sw.stop();    System.***out***.println(sw.nanoseconds());  }  }    @Test  **public** **void** ExecutionTimeArraySort()  {    **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    Arrays.*sort*(expdata);  sw.stop();    System.***out***.println(sw.nanoseconds());  }  }      //@Test  **public** **void** FindOptimalKValue()  {      **long**[] nanosec = **new** **long**[100];  **for** (**int** j = 0; j < 100; j++) {  **int** KVal = 0;  **for** (**int** i = 0; i < 100; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  QuickSortFixedPivot sorta = **new** QuickSortFixedPivot();  sorta.setAlg(**new** InsertionSort());    sorta.K = KVal;    Data dt = **new** Data(10000, Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    sorta.sort(expdata);  sw.stop();    **if**(nanosec[KVal] < sw.nanoseconds())  nanosec[KVal] = sw.nanoseconds();  //System.out.println( sorta.K +"\t"+ sw.nanoseconds());    KVal++;  }  }  **for** (**int** i = 0; i < nanosec.length; i++) {  System.***out***.println(i +"\t" + nanosec[i] );  }  }    //@Test  **public** **void** DifferentDataSetsRandom()  {  **int**[] randomarray = **new** **int**[]{1,2,3,4,5};  InsertionSort sorter = **new** InsertionSort();    sorter.sort(randomarray);  Arrays.*sort*(randomarray);  // doing things above to minimise java just in time...    **for** (**int** i = 0; i <= 18; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  Data dt = **new** Data((**int**)Math.*pow*(2, i), Integer.***MAX\_VALUE***, Order.***RANDOM***);    **int**[] expdata = dt.get();    Arrays.*sort*(expdata);  sw.stop();      **long** sortArrays = sw.nanoseconds();    sw.start();  expdata = dt.get();  sorter.sort(expdata);  sw.stop();    System.***out***.println(sortArrays + "\t" + sw.nanoseconds());  }    /\*  System.out.println("break");  for (int i = 0; i <= 25; i++) {  Stopwatch sw = new Stopwatch();  sw.start();  //ALGORITHM 1  Data dt = new Data((int)Math.pow(2, i), Integer.MAX\_VALUE, Order.RANDOM);    int[] expdata = dt.get();    sorter.sort(expdata);  sw.stop();    System.out.println(sw.nanoseconds());  }\*/  }    @Test  **public** **void** DifferentDataSetsOrdered()  {  **int**[] randomarray = **new** **int**[]{1,2,3,4,5};  InsertionSort sorter = **new** InsertionSort();    sorter.sort(randomarray);  Arrays.*sort*(randomarray);  // doing things above to minimise java just in time...    **for** (**int** i = 0; i <= 18; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  Data dt = **new** Data((**int**)Math.*pow*(2, i), Integer.***MAX\_VALUE***, Order.***ASCENDING***);    **int**[] expdata = dt.get();    Arrays.*sort*(expdata);  sw.stop();      **long** sortArrays = sw.nanoseconds();    sw.start();  expdata = dt.get();  sorter.sort(expdata);  sw.stop();    System.***out***.println(sortArrays + "\t" + sw.nanoseconds());  }  }      @Test  **public** **void** DifferentDataSetsDiscending()  {  **int**[] randomarray = **new** **int**[]{1,2,3,4,5};  InsertionSort sorter = **new** InsertionSort();    sorter.sort(randomarray);  Arrays.*sort*(randomarray);  // doing things above to minimise java just in time...    **for** (**int** i = 0; i <= 18; i++) {  Stopwatch sw = **new** Stopwatch();  sw.start();  //ALGORITHM 1  Data dt = **new** Data((**int**)Math.*pow*(2, i), Integer.***MAX\_VALUE***, Order.***DESCENDING***);    **int**[] expdata = dt.get();    Arrays.*sort*(expdata);  sw.stop();      **long** sortArrays = sw.nanoseconds();    sw.start();  expdata = dt.get();  sorter.sort(expdata);  sw.stop();    System.***out***.println(sortArrays + "\t" + sw.nanoseconds());  }  }  } |

## Sorting algorithms

### ISort

|  |
| --- |
| **package** inda6;  **import** java.util.function.BiConsumer;  **public** **interface** ISort {  /\*\*  \* Sorts the array into ascending numerical order.  \*/  **void** sort(**int**[] v);    **void** specsort(**int**[] v, **int** first, **int** last);    //void setAlg(ISort sortAlg);    **void** setAlg(ISort alg);  } |

### Insertion Sort

|  |
| --- |
| **package** inda6;  **public** **class** InsertionSort **implements** ISort {    /\*\*  \* Insertion sort from Inda 2014.  \* **@param** v  \*/  **public** **void** sort(**int**[] v)  {  specsort(v, 0, v.length);  }  **public** **void** specsort(**int**[] v, **int** first, **int** last)  {  **for** (**int** j = first + 1; j < last; j++)  {  **int** key = v[j];  **int** i = j-1;    **while**(i>= first && v[i] > key)  {  v[i+1] = v[i];  i--;  }  v[i+1] = key;  }  }    **public** **void** setAlg(ISort sort)  {    }  } |

### Quick Sort – fixed pivot

|  |
| --- |
| **package** inda6;  **public** **class** QuickSortFixedPivot **implements** ISort {    **private** ISort sortingAlg;  **public** **int** K = 10;    **public** **void** sort(**int**[] v)  {  specsort(v, 0, v.length-1);  }  **public** **void** specsort(**int**[] v, **int** first, **int** last) {    **if** (first >= last) // Less than two elements  **return**;    **if** (last-first < K)  {  //perform insertion sort.  //InsertionSort sorter = new InsertionSort();  **if**(sortingAlg != **null**)  sortingAlg.specsort(v, first, last);    }    // Choose a pivot element.  **int** p = v[first];  // Partition the elements so that every number of  // v[first..mid] <= p and every number of v[mid+1..last] > p.  **int**[] mid = partition(v, first, last, p);  specsort(v, first, mid[0]);  specsort(v, mid[1], last); // removed +1  }    **public** **int**[] partition(**int**[] v, **int** first, **int** last, **int** pivot) {  **int** low = first;  **int** mid = first;  **int** high = last;  **while** (mid <= high) {  // Invariant:  // - v[first..low-1] < pivot  // - v[low..mid-1] = pivot  // - v[mid..high] are unknown  // - v[high+1..last] > pivot  //  // < pivot = pivot unknown > pivot  // -----------------------------------------------  // v: | | |a | |  // -----------------------------------------------  // ^ ^ ^ ^ ^  // first low mid high last  //  **int** a = v[mid];  **if** (a < pivot) {  v[mid] = v[low];  v[low] = a;  low++;  mid++;  } **else** **if** (a == pivot) {  mid++;  } **else** { // a > pivot  v[mid] = v[high];  v[high] = a;  high--;  }  }  **return** **new** **int**[]{low,mid};  }    @Override  **public** **void** setAlg(ISort alg) {  **this**.sortingAlg = alg;    }  } |

### Quick Sort – random pivot

|  |
| --- |
| **package** inda6;  **import** java.util.Random;  **public** **class** QuickSortRandPivot **implements** ISort {    **private** ISort sortingAlg;  **public** **int** K = 10;    **public** **void** sort(**int**[] v)  {  specsort(v, 0, v.length-1);  }  **public** **void** specsort(**int**[] v, **int** first, **int** last) {    **if** (first >= last) // Less than two elements  **return**;    **if** (last-first < K)  {  //perform insertion sort.  //InsertionSort sorter = new InsertionSort();  **if**(sortingAlg != **null**)  sortingAlg.specsort(v, first, last);    }    // Choose a pivot element.  Random rn = **new** Random();  **int** p = v[rn.nextInt(last-first) + first];  // Partition the elements so that every number of  // v[first..mid] <= p and every number of v[mid+1..last] > p.  **int**[] mid = partition(v, first, last, p);  specsort(v, first, mid[0]);  specsort(v, mid[1], last); // removed +1  }    **public** **int**[] partition(**int**[] v, **int** first, **int** last, **int** pivot) {  **int** low = first;  **int** mid = first;  **int** high = last;  **while** (mid <= high) {  // Invariant:  // - v[first..low-1] < pivot  // - v[low..mid-1] = pivot  // - v[mid..high] are unknown  // - v[high+1..last] > pivot  //  // < pivot = pivot unknown > pivot  // -----------------------------------------------  // v: | | |a | |  // -----------------------------------------------  // ^ ^ ^ ^ ^  // first low mid high last  //  **int** a = v[mid];  **if** (a < pivot) {  v[mid] = v[low];  v[low] = a;  low++;  mid++;  } **else** **if** (a == pivot) {  mid++;  } **else** { // a > pivot  v[mid] = v[high];  v[high] = a;  high--;  }  }  **return** **new** **int**[]{low,mid};  }    @Override  **public** **void** setAlg(ISort alg) {  **this**.sortingAlg = alg;    }  **public** **void** setK(**int** k)  {  **this**.K = k;  }  } |