# Maximizing Performance with Bespoke Programming

By Adrian Marriott, an independent consultant at Logos Software.

# Introduction

Commercial systems are developed with huge range of performance requirements and we are concerned in this paper with that small number of systems where absolutely maximum performance is demanded; either in terms of execution speed or of available memory. We discuss the role of ***bespoke implementation*** and show that writing programs that utilize novel data structures and new algorithms designed with knowledge of the specific problem context is a necessary complement to the generic components and automatic optimizations offered by compilers and modern JVMs in order to maximize performance.

## *Empirical Tests*

We demonstrate this thesis here with an empirical comparison of the standard Java sort routines working with integers and bespoke implementations optimized to handle integers in specific contexts. Graphs comparing the speed of these algorithms are presented.

All the empirical tests described in this paper are run several times and an average taken, so we eliminate any random variations introduced by unexpected operating system activity. These tests were run on an isolated system, with no other users or applications operating, to minimize skewing of results. The results are output directly by the benchmark programs to file, outside the timed code section, and the results are correlated by an automatic graphing program[1] to remove the possibility of human error. The correlation program also checks that every average calculated for each point on the graphs is calculated from the same number of underlying samples, and reports the standard deviation and identifies any outliers. These are either, ignored because they are not significant, or the entire benchmark can be re-run.

# Bespoke Implementation

## *Efficient Code*

The central proposal here is not new or controversial and has been clearly communicated in computing literature for many years [1] that as a rule the fewer instructions a program executes the faster it runs. This is generally true of programs written in any language, and is therefore also true of programs written in Java and C++, both common language choices for large, scalable, fast, distributed systems.

Pragmatically this requires hand-crafted code written in a general programming language such as Java or C++, that accounts for a specific programming context, to successfully implement the fastest, most memory efficient programs. In contrast relying solely on configuring generic components, designed and implemented without knowledge of the

[1] This java utility is available free at: http://www.logos-software.com

specific circumstances of their use, whether these be off-the-shelf libraries or the automatic optimization of compilers, even with advances in JVM technology [2][3], might be sufficient in many circumstances but may not necessarily be optimal. Ultimately, bespoke design, implementation and optimization, and the creative skill of the programmer used *to complement* these generic components will produce the fastest, most scalable programs.

## Generic Sort vs. Counting Sort

We demonstrate this here with an empirical comparison of the standard Java sort routines with a bespoke implementation designed with a specific context in mind. The first use-case is simply to sort ***n*** integers each of which has a positive value between zero and a maximum value ***k***. In Java this can be implemented easily using either the `Collections` or the `Arrays` class.

```
/**
 * Sorts the input sequence using Collections sort
 */
public ArrayList<Integer> sortInts(ArrayList<Integer> inputSequence)
{
      Collections.sort(inputSequence);
      return inputSequence;
}


/**
 * Sorts an array of integers using the Arrays class
 */
public int[] sortInts(int [] inputSequence)
{
      Arrays.sort(inputSequence);
      return inputSequence;
}
```

The java documentation describes the `Collections.sort` routine.

*"This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort. … The sorting algorithm is a modified mergesort [2] (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n log(n) performance. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the $n^2$ log(n) performance that would result from attempting to sort a linked list in place."*

The `Collections.sort` routine has been designed generically to support any element type, and therefore the sort algorithm used cannot exploit the particular features of our

[2] For details on the merge sort see: http://en.wikipedia.org/wiki/Merge_sort

case, specifically to sort positive integers. The documentation for the `Arrays.sort` routine describes a more integer-specific algorithm

*"Sorts the specified array of ints into ascending numerical order. The sorting algorithm is a tuned quicksort, adapted from Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993).[3] This algorithm offers n*log(n) performance on many data sets that cause other quicksorts to degrade to quadratic performance."*

The `Arrays.sort` algorithm has been designed to sort integers, and is much closer to our specific requirements, so we would expect this to run relatively efficiently. However, the *O(n log(n))* performance mentioned is higher than the best possible for non-comparative sort routines, so there may be a way to improve on this.

Now, if we consider a sort routine optimized for positive integers, then we can note the following facts true of integers, but not necessarily true of any arbitrary element type:

1. Integers are not dense on the number line, unlike real or floating point numbers. Given two integers *(a, b)* that differ by one *(a+1 = b)*, it is impossible to find a third integer *(x)* between them such that $a < x < b$.

2. There is no associated data with these integers; they are not tuples. So the actual identity of the input elements can be eliminated in the sort process if this leads to an efficiency saving.

The points above act as constraints on our input sequence, and it is possible to exploit these to write a very different sort routine, a variant of the Counting Sort[4], as shown below.

```
/**
 * Implements the counting sort algorithm
 */
public int[] countingSort(int[] inputSequence)
{
	// Find the maximum value
	int maxValue = -1;
	for(int i = 0; i < inputSequence.length; i++)
	{
		int x = inputSequence[i];
		if(x > maxValue)
		{
			maxValue = x;
		}
	}

	// Allocate an array, 1 for each value
	int[] counts = new int[maxValue + 1];
```

[3] http://www.enseignement.polytechnique.fr/informatique/profs/Luc.Maranget/421/09/bentley93engineering.pdf
[4] For details on the counting sort see: http://en.wikipedia.org/wiki/Counting_sort

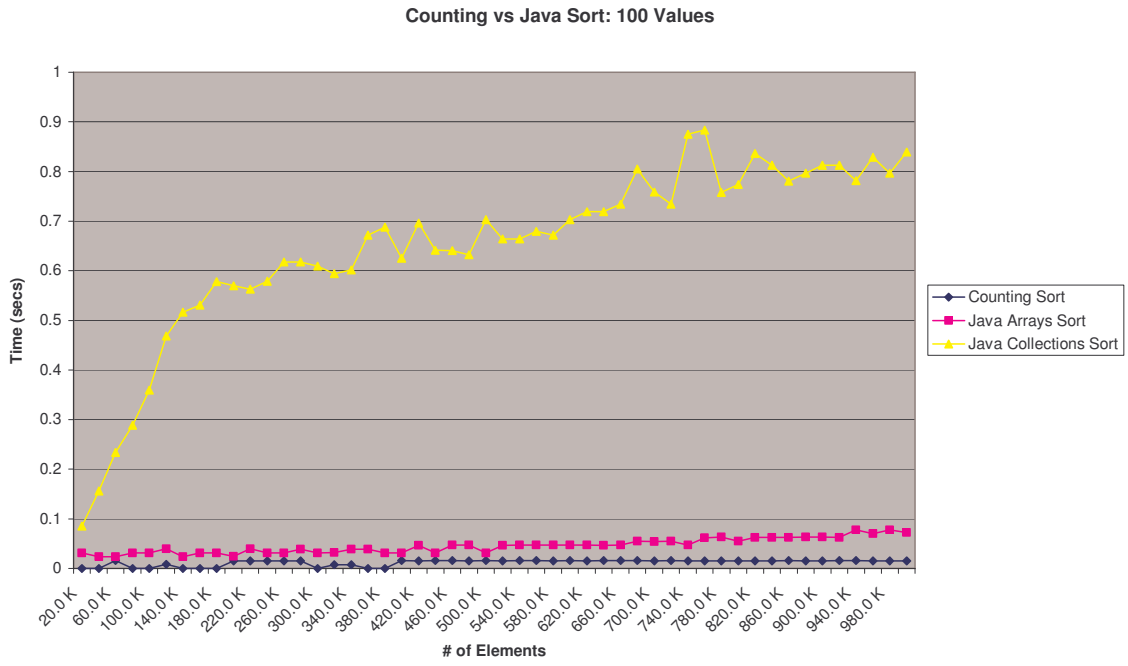
```
        // Count the occurrences of each value
        for(int i : inputSequence)
        {
              counts[i] += 1;
        }

        // Spin through re-writing the counted values in order
        for(int i = 0, j = 0; i <= maxValue; i++)
        {
              int c = counts[i];
              for(int k = 0; k < c; k++, j++)
              {
                    inputSequence[j] = i;
              }
        }

        return inputSequence;
}
```

This routine starts by finding the maximum value in the input sequence, which it uses to size an array of `counts`. The input sequence is then examined again and the number of occurrences of each value is accumulated at the corresponding offset in the `counts` array. For example, an input sequence of `[3,1,4,7,1,4,0]` would result in a `counts` array of length 8, containing the following values `[1,2,0,1,2,0,0,1]`. Finally the code spins through the `counts` array and *overwrites* the `inputSequence` with the values in sorted order. In our example this gives `[0,1,1,3,4,4,7]`.

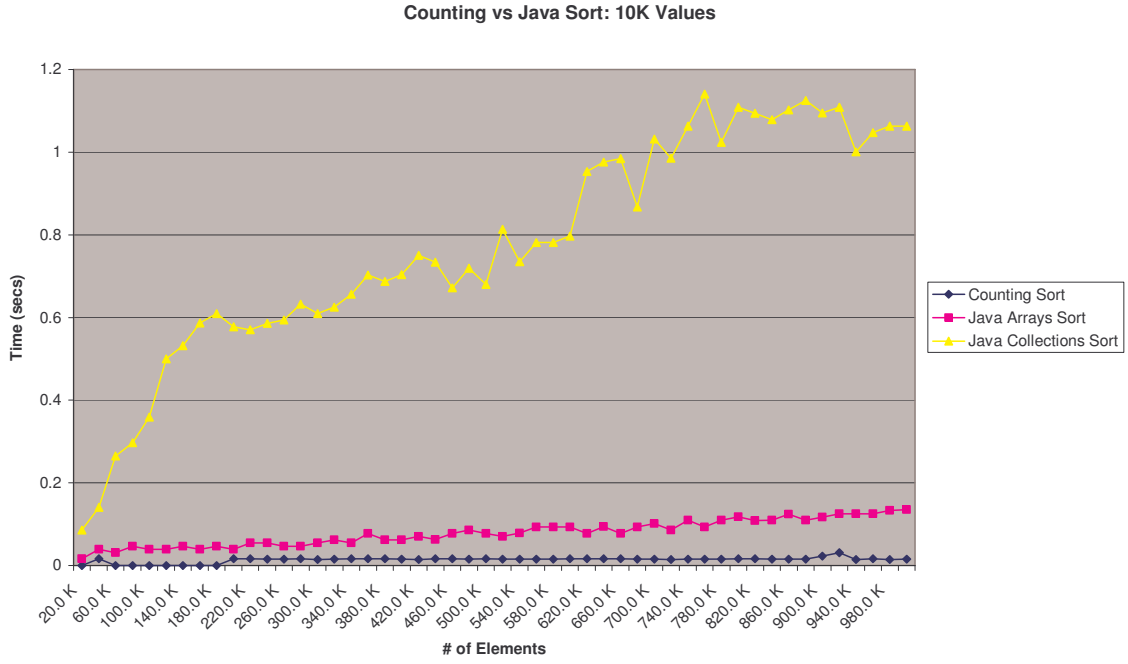
This clarifies why this algorithm cannot possibly sort real or floating point numbers; because finding the maximum in an input sequence of floats cannot tell us how large to allocate the `counts` array, such that there is a slot for each possible float value in the input sequence; nor is there any way, of mapping successive floating point numbers to the correct index in the `counts` array. It's also clear that this counting sort cannot work with tuples with integer keys, because the original identity of the elements is destroyed when the last step in the algorithm overwrites the original input elements.

This variant of the counting sort algorithm passes through the input sequence three times; once to find the maximum value (i.e. the value ***k***) and this operation is $O(n)$ . It allocates an array of size ***k*** and even though this is always a single array allocation this operation is proportional to the size allocated, $O(k)$. It then passes through the input sequence a second time as it accumulates the counts of individual values. Finally it spins through the input sequence overwriting the contents with the sorted values, proportional to $O(n)$. This gives an overall time complexity of $O(n+k)$ and memory usage proportional to $O(k)$. This counting sort is therefore sensitive not only to the number of elements sorted, but also the range of values within the input sequence.

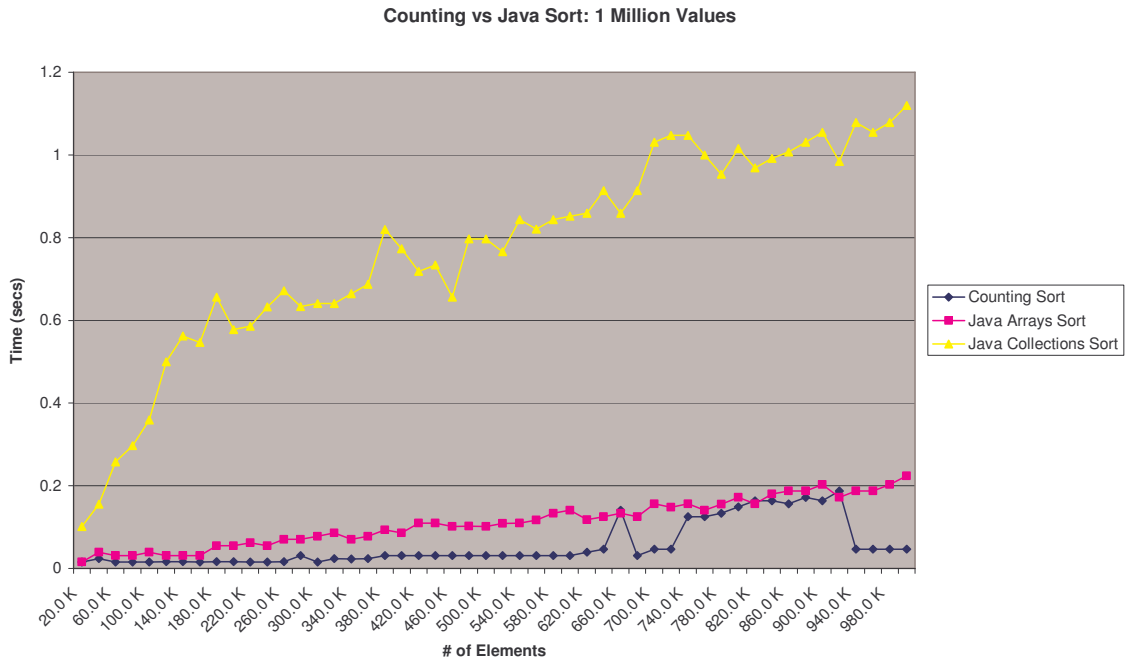
Below we show graphs comparing the three sorting routines for input sequences of different sizes and different ranges of values. Note in every test run, all the algorithms had identical pseudo-random number sequences to sort, so all the routines had identical amounts of work to do.




Here we see a graph showing the time taken to sort integers with a range of values between 0 and 100 (i.e. ***k***=100) for with the number of elements, ***n*** between 20 thousand and 1 million.




Above we show results for the same test but this time with the range of values between 0 and 10 thousand (i.e. ***k***=10,000).




Lastly, the results with $k$=1 Million. These results show the counting sort is consistently faster than both the algorithms used in the Java libraries, over all the ranges of $n$ between 20 thousand and 1 million, and all ranges of $k$ between 100 and 1 million. The degree of improvement increases as $n$ and $k$ increase, up to a factor of 4.7 faster than the `Arrays` sort and a factor of 23.8 faster than the `Collections` sort. However, simply sorting integers alone is not particularly useful. Next we expand our example by sorting integer tuples.

## *Generic Sort vs. Pigeon-Hole Sort*

Here we relax one of the constraints on our input sequence by allowing arrays of tuples, which means we have to preserve the identity of the elements in the input sequence. In this scenario we need to sort $n$ ordered pairs of integers the first of which acts as a key value between zero and a maximum value $k$. The second integer in the pair plays the role of 'associated data', and only serves to require our routine to preserve the identity of each input element. Below we show a Java implementation using the `Collections` and the `Arrays` classes.

```
public interface PigeonHoleable
{
	/**
	 * Returns the integer key
	 */
	int getKey();
}

public class OrderedPair implements PigeonHoleable
{
	private int k;
	private int x;
```

```
	public OrderedPair(int k, int x)
	{
		this.k = k;
		this.x = x;
	}

	public int getKey()
	{
		return k;
	}

	public String toString()
	{
		return "(" + k + "," + x + ")";
	}
}
```

Above we see the `OrderedPair` class, containing two integers, one of which, `k`, is the key on which sequences of these objects will be sorted. The interface expected by our sort routines is `PigeonHoleable`.

```
/**
 * Sorts a list of ordered pairs using java Collections sort
 */
public ArrayList<PigeonHoleable>
	pairSort(ArrayList<PigeonHoleable> inputSequence)
{
	Collections.sort(inputSequence, new Comparator<PigeonHoleable>(){
		public int compare(PigeonHoleable o1,PigeonHoleable o2)
		{
			return (o1.getKey() - o2.getKey());
		}
	});

	return inputSequence;
}

/**
 * Sorts an array of integers pairs using the Arrays class sort
 */
public PigeonHoleable[]
	pairSort(PigeonHoleable[] inputSequence)
{
	Arrays.sort(inputSequence, new Comparator<PigeonHoleable>(){
		public int compare(PigeonHoleable o1,PigeonHoleable o2)
		{
			return (o1.getKey() - o2.getKey());
		}
	});

	return inputSequence;
}
```

Here we have functions which will sort these ordered pairs using the `Collections` and `Arrays` sort methods as before, but this time using a comparator which compares the keys.

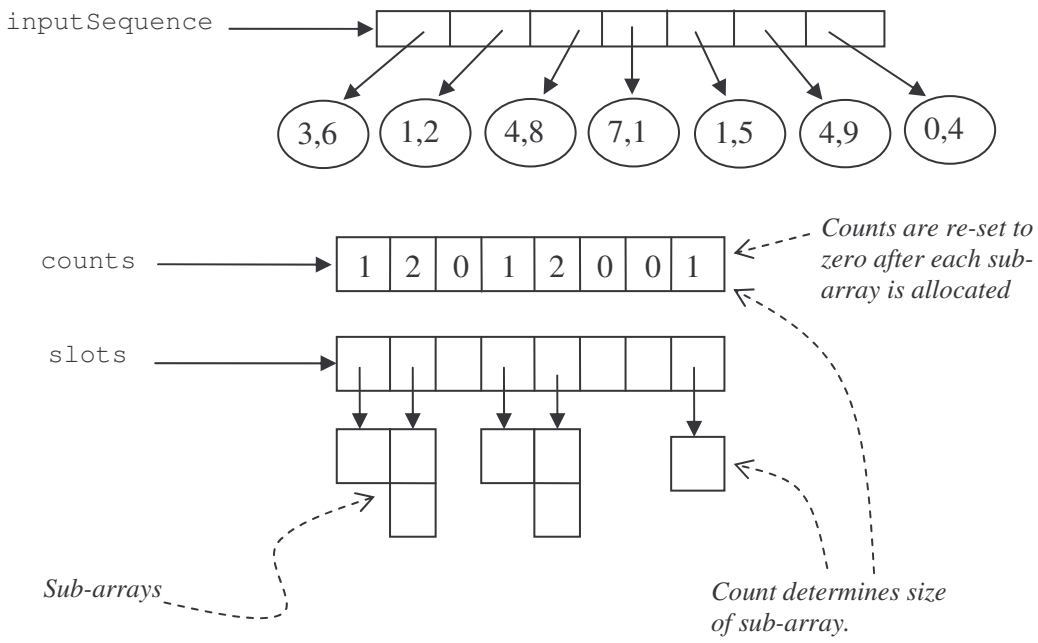
The bespoke code consists of a variant of the Pigeon-Hole Sort[5], the code for which is shown below.

```
/**
 * Implements the pigeon-hole sort algorithm, and maintains
 * the identity of the elements, so it can deal with other types
 * than integers. This means that whole records can be sorted
 * effectively, with the associated fields remaining with the
 * original element. This is a stable sort.
 */
public PigeonHoleable[] pigeonHoleSort(PigeonHoleable[] inputSequence)
{
      // Find the maximum value
      int maxValue = -1;
      for(int i=0; i<inputSequence.length; i++)
      {
            int x = inputSequence[i].getKey();
            if(x > maxValue)
            {
                  maxValue = x;
            }
      }

      // Allocate an array, 1 for each value
      int[] counts = new int[maxValue + 1];

      // Count the occurences of each key value
      for(PigeonHoleable b : inputSequence)
      {
            counts[b.getKey()] += 1;
      }

      // Allocate a ragged array of arrays
      PigeonHoleable[][] slots = new PigeonHoleable[maxValue + 1][];
      for(int i=0; i<counts.length; i++)
      {
            int size = counts[i];
            if(size>0)
            {
                  slots[i] = new PigeonHoleable[size];
            }

            // Zero the counts array as we go so we can reuse them
            // for indexes
            counts[i]=0;
      }

      // Now spin through the input inserting elements directly
      // into the correct slots
```

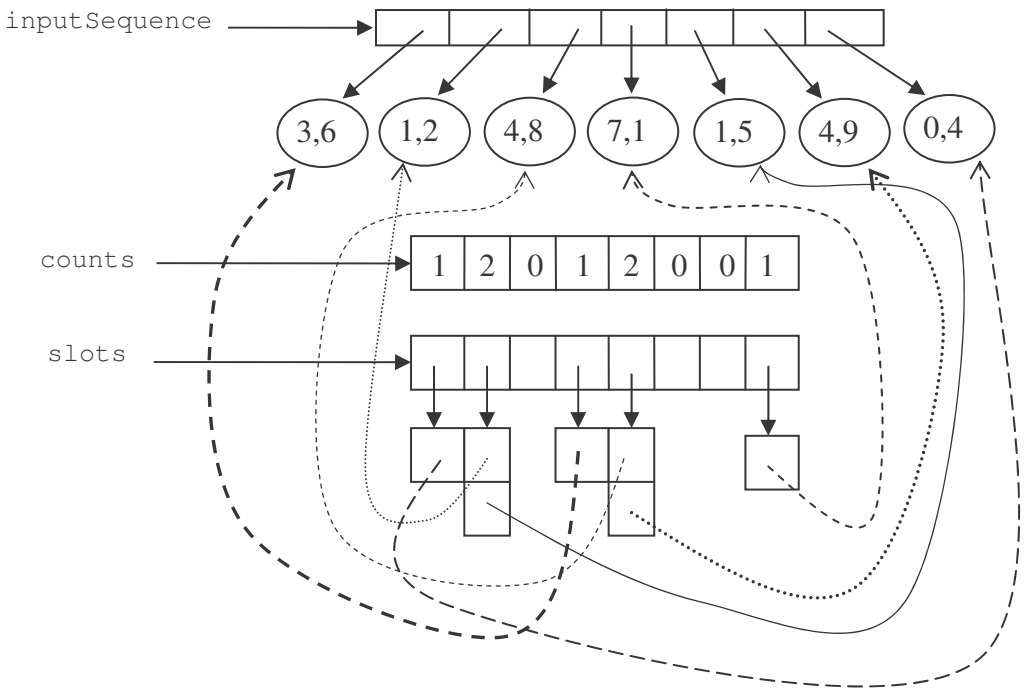
[5] For details on the Pigeon-Hole sort see: http://en.wikipedia.org/wiki/Pigeonhole_sort

```
        for(int i=0; i<inputSequence.length; i++)
        {
              PigeonHoleable bs  = inputSequence[i];
              int k = bs.getKey();
              int j = counts[k]++;
              slots[k][j]=bs;
        }

        // Spin through re-inserting the slot references back into
        // the input sequence in the correct order
        for(int i = 0, j = 0; i <= maxValue; i++)
        {
              int c = counts[i];
              for(int k = 0; k < c; k++, j++)
              {
                    inputSequence[j] = slots[i][k];
              }
        }

        return inputSequence;
}
```

In a similar way to the counting sort, this routine starts by finding the maximum key value in the input sequence, which it uses to size an array of `counts`. The input sequence is then examined again and the number of occurrences of each key value is accumulated at the corresponding offset in the `counts` array. Next the totals in the `counts` array are used in a loop to size a ragged array-of-arrays called `slots`, and as each sub-array is allocated the corresponding count is reset back to zero. This is illustrated below.



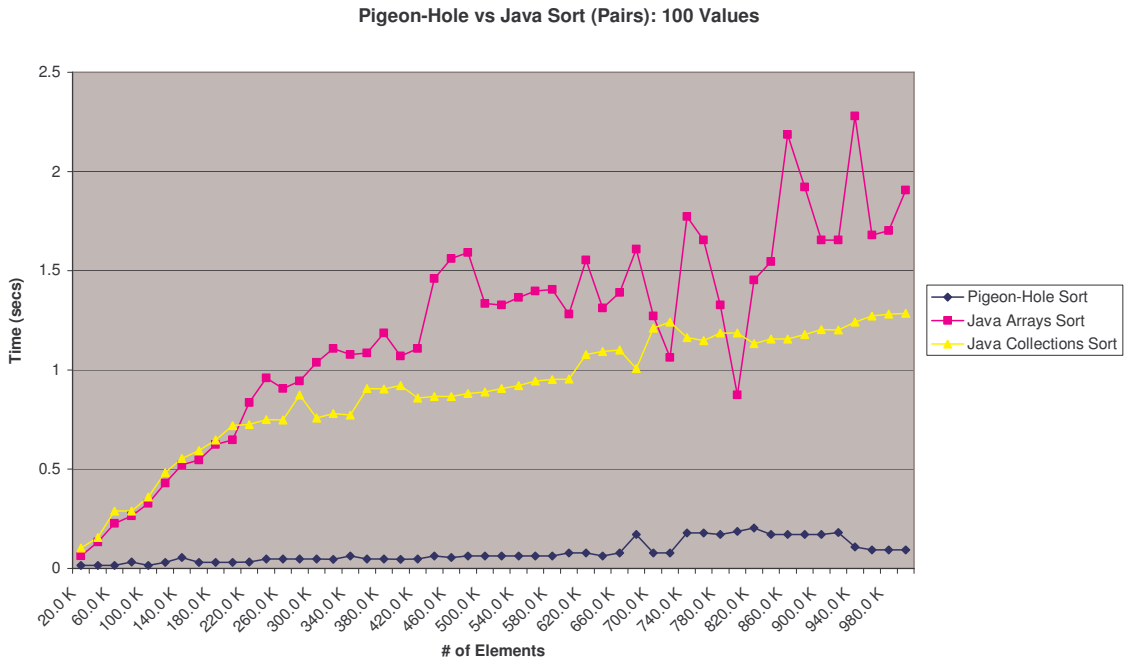
Then the code spins through the input sequence and inserts a reference to each input element directly into the slots array, using the key value as the first index, and the value in the counts array as the second index. In this loop the counts array has been 're-purposed' and holds the 'next index'. This is why each of the counts is reset to zero. This stage is illustrated below.



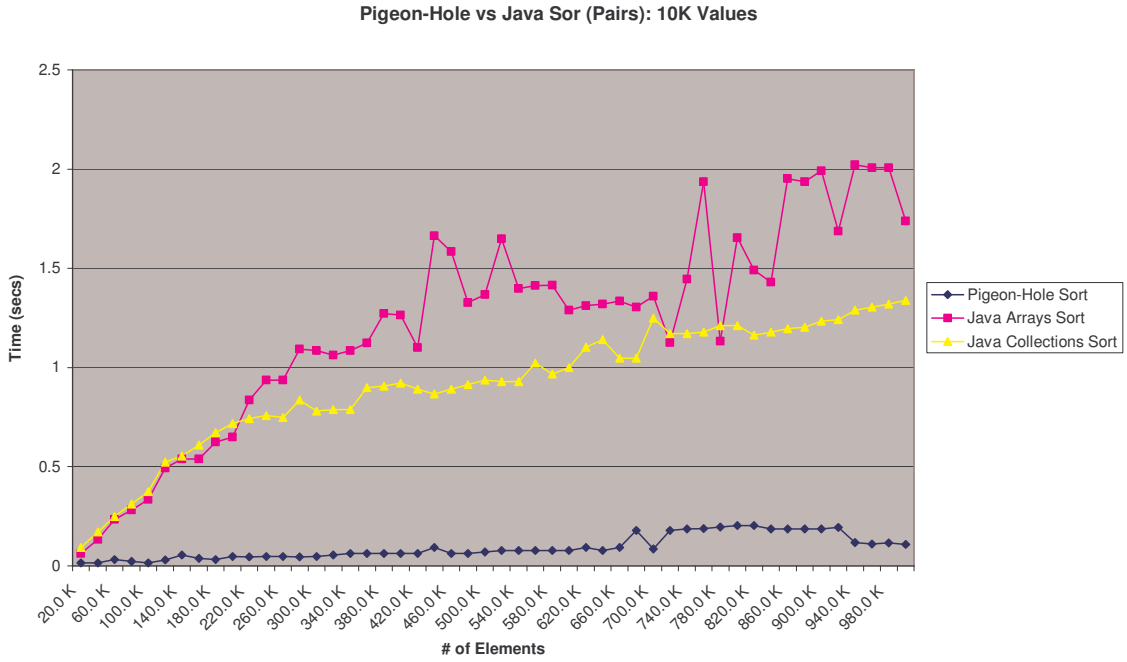

Finally the ragged two-dimensional slots array is navigated and the references are read out in sorted order. So the original input sequence `(3,6)(1,2)(4,8)(7,1)(1,5)(4,9)(0,4)` is sorted into `(0,4)(1,2)(1,5)(3,6)(4,8)(4,9)(7,1)`.

This implementation of the pigeon-hole sort algorithm passes through the input sequence four times; all of which take a time proportional to $O(n)$. Once to find the maximum value (the value $k$) , a second time to accumulate the counts of individual values, a third time inserting references directly into the sub-arrays and finally it iterates across all the sub-arrays copying the references back into the input sequence in sorted order. In terms of space, this algorithms allocates two arrays, `counts` and `slots`, of size $k$, and a set of sub-arrays with one slot for each element in the input sequence, a space proportional to $O(n)$. This gives an overall time complexity and memory usage proportional to $O(n+k)$. Like the counting sort described previously, this pigeon-hole sort is a stable sort and sensitive not only to the number of elements sorted, but also the range of values within the input sequence.

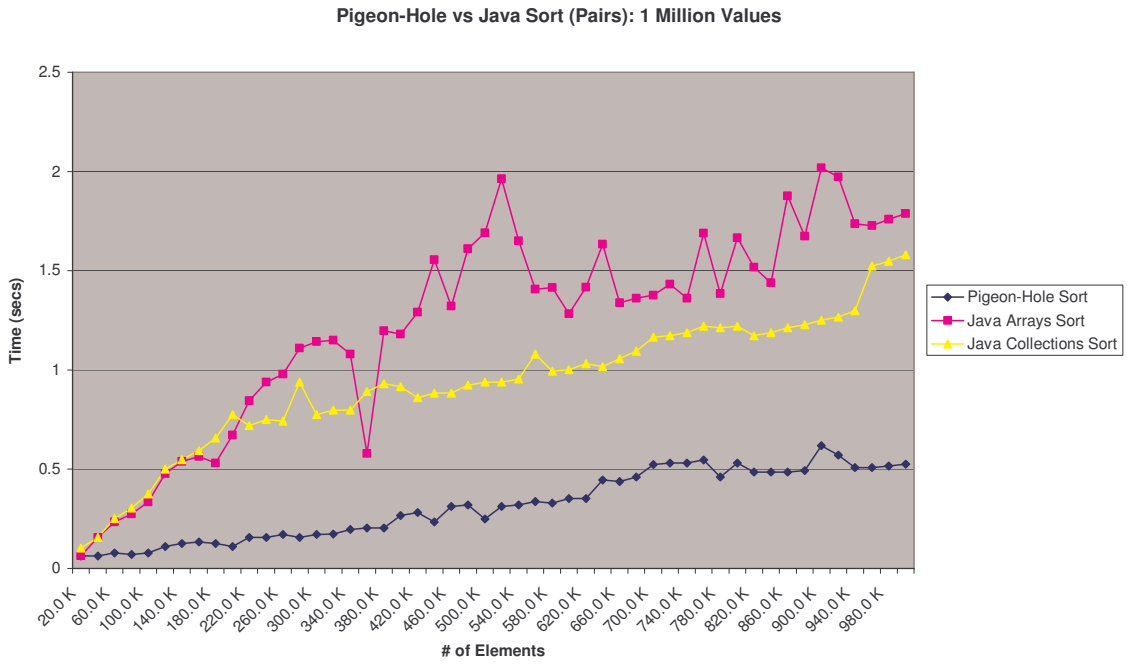
Now we show graphs comparing the performance of the three sorting routines for input sequences of ordered pairs of integers, with different numbers of elements ($n$) and different ranges of values ($k$).




Here we see a graph showing the time taken to sort ordered pairs with a range of key values between 0 and 100 (i.e. ***k***=100) for with the number of elements, ***n*** between 20 thousand and 1 million.




Above we show results for the same test but this time with the range of key values between 0 and 10 thousand (i.e. ***k***=10,000).




Lastly, the results with ***k***=1 Million. These results show the pigeon-hole sort is consistently faster than both the algorithms used in the Java libraries, over all the ranges of ***n*** between 20 thousand and 1 million, and all ranges of ***k*** between 100 and 1 million.

The performance of the java `Arrays` sort with ordered pairs is significantly worse than when sorting integers, whereas the `Collections` sort is remarkably consistent, and is actually faster than the `Arrays` implementation. However, both are much slower than the pigeon-hole sort, particularly with low values of ***k***, where the degree of improvement over the `Arrays` sort is a between 15 and 17 times faster, and over the `Collections` sort by a factor of 12 or 13. When the value of ***k*** gets larger, in the range of 1 million, this improvement drops to a factor of 3 or 4, probably because the time taken to allocate memory is becoming significant.

Further efficiency improvements could probably be made by pre-allocating the memory necessary to execute the sort and reusing it between invocations, but the feasibility of this might depend on additional context specific knowledge.

## Conclusion

There are provably an infinite number of ways to implement any given program[6]. If we assume a performance requirement for the shortest execution time, and assume that each instruction executes in the same amount of time[7], the program executing the fewest instructions will run the quickest. There may be more than one optimal program, and many near-optimal programs.

These optimal programs can either be written entirely by hand, an impractical proposition, or automatic optimization technologies, such as optimizing compilers, can be employed to analyze the program and generate efficient code. Automatic optimization technologies have traditionally analyzed the static structure of the code [4][5] but modern Java environments are also starting to account for dynamic program behaviour by collecting statistics during execution, using techniques such as Escape Analysis [6], and then dynamically recompiling executing code on-the-fly [7]. In either case, only information that is *explicit* within the program, or such as may be logically deduced from its behaviour, can be utilized by the optimization decision procedure. Some optimizations are impossible or impractical to make automatically, because knowledge *specific to the context of use* is required, and this information is not explicitly encoded in the program, or reliably derivable from its behaviour.

For example, take a program that is reading event objects from a socket in batches, and sorting them on a particular integer key field, and then inserting these sorted batches into another in-memory index structure. If this code is written so the key field is read as an integer data type, which is necessary to produce the correct sort order, but these objects are passed to sort routines which are designed to work with arbitrary data types, including strings and floats, then the sort processing is sub-optimal. Conversely, if the events are processed by a sort routine specifically designed to handle integers efficiently, and for a particular range of values, the sort will run significantly faster. If the sort processing constitutes a significant proportion of the overall processing, then this optimization will have a considerable impact on overall system performance and scalability.

In this example, it is the fact that we are sorting integer sequences with particular values of $\boldsymbol{n}$ and $\boldsymbol{k}$ that comprise the specific context of use. We have then designed and written sort routines which utilize this context specific information to improve the performance compared to generic library code. In our case, it would be very easy to supply this code in a Java library, as a sort function overloaded to process integer sequences, for 3rd parties to

[6] For example, a proof by induction where spurious instructions are inserted into the program to increment the value $\boldsymbol{v}$ in any memory location by an arbitrary amount $\boldsymbol{x}$, and then decrement it back to its previous value $\boldsymbol{v}$ before proceeding. Since the size of $\boldsymbol{x}$ can be of any magnitude, and the final output of the program is unaffected, there are an infinite number of implementations of any program.

[7] Clearly this is untrue for real instruction sets, for example compare the time to execute the `mov` or `add` instructions with `stosd` on the Intel 8086 processor family, but as stated here this is a simplified form of an argument that can be restated for real instruction sets, where each instruction type runs in different times. Both arguments are valid given their assumptions, and both are relevant to and support the subject under discussion.

reuse under similar circumstances to our own, but this does not detract from the point. No amount of automatic optimization, code analysis or dynamic recompilation could produce the same results given the choice of sort routines available in the Java library. Genuinely new code was required, using a novel algorithm and new data structures, which exploited knowledge specific to the context of use.

# References

[1] Programming Perls, by Jon Bentley, ACM Press, ISBN-13: 978-0201657883
http://www.amazon.co.uk/Programming-Pearls-ACM-Press-Bentley/dp/0201657880/ref=sr_11_1/203-2385316-8763101?ie=UTF8&qid=1185469929&sr=11-1

[2] The Jikes Research Virtual Machine (RVM),
http://www-128.ibm.com/developerworks/java/library/j-jalapeno/

[3] Jikes RVM source code download from SourceForge.net,
http://sourceforge.net/project/showfiles.php?group_id=128805&package_id=141063&release_id=498110

[4] GCC Optimization Options, http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

[5] Optimizations in GCC, by M. Tim Jones, Linux Journal, http://www.linuxjournal.com/article/7269

[6] Escape Analysis for Java, by J. Choi, M. Gupta M. Serrano V.C. Sreedhar S. Midkiff, IBM T. J. Watson Research Center, (1999) http://www.research.ibm.com/people/g/gupta/escape.ps

[7] Dynamic Compilation and Performance Measurement, by Brian Goetz, Dec 2004.
http://www-128.ibm.com/developerworks/library/j-jtp12214/