Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 225 (2009) 391–404
www.elsevier.com/locate/entcs

Sorting Algorithms in *MOQA*

Jacinta Townley[1,3] Joseph Manning[2,4] Michel Schellekens[2,5]

CEOL Research Group
Dept. Computer Science
University College Cork
Cork, Ireland

Abstract

A high-level overview of the *MOQA* language is presented. The representation of its data structure, a labeled series-parallel partial order, is shown along with some of the functions allowed upon this data structure. The combination of *MOQA*'s data structure and its functions capture the required calculus to statically obtain the average-case time of algorithms written in this language. The implementation of these concepts is discussed and algorithms written in this implementation are presented. A detailed analysis of one of these implemented algorithms, quicksort, critically compares its average-case time in *MOQA* against the average-case time of standard quicksort. While the asymptotic average of quicksort in *MOQA* remains unchanged, extra constant costs are incurred by the *MOQA* method. It is shown that these costs result from molding the algorithm around the *MOQA* data structure and functions versus the general approach of choosing the data structure and functions that best match the algorithm. This limitation is balanced against an approach that aims to obtain the average-case time of algorithms statically.

Keywords: partial order, analysis of algorithms, average case, randomness preservation, sorting, quicksort

1 Introduction

The *MOQA*(MOdular Quantitative Analysis) language is implemented on the concepts developed by [18,21], which aim to automatically derive the average-case time of an algorithm by the randomness-preservation of the algorithm's data throughout its execution. The randomness-preservation of data means informally that data is represented and controlled in a manner that tracks its distribution at all times, removing all uncertainty about the possible states of the data at any moment during its lifetime. The data is controlled by *MOQA* functions and when a program is written with these functions, with certain constraints on the control flow of the program,

[1] Research supported by Science Foundation Ireland under Grant 05/RFP/CMS0044
[2] Research supported by Science Foundation Ireland under Grant 02/IN.1/181
[3] Email: j.townley@cs.ucc.ie
[4] Email: manning@cs.ucc.ie
[5] Email: m.schellekens@cs.ucc.ie

doi:10.1016/j.entcs.2008.12.088

and then statically analysed, all possible input and outputs states for each *MOQA* function can be tracked along with the probability of each state occurring. This is then used to produce the average-case recurrence for the algorithm in question.

The following section, Section 2, covers related research. Section 3 will give a concise explanation of the *MOQA* data structure and some of the main functions allowed upon this data structure. This description will be brief as it is not the purpose of this paper to reiterate work already described in [18,21] and it is not necessary to have a deep understanding of the mathematical concepts behind *MOQA* for the scope of this paper. Section 4 will be an overview of the current state of the *MOQA* implementation and Section 5 will discuss the overhead of two algorithms written in this implementation. The two algorithms are insertion-sort and quicksort. In Section 5 particular attention is given to the fact that *MOQA* specifies the relation between data in its structures instead of the algorithms specifying what the relation is. Section 5 will show that the impact of this is a higher space cost and for the average-case time of quicksort, an increase in constant values. This paper will then conclude with a summary, Section 6, and an outline of future work in this area, Section 7.

2 Related Research

Much research has been done in the area of static analysis. To track particular program behaviours statically, a wide variety of approaches have been developed. The behaviour of interest to *MOQA* is tracking the program's data structures. In [2], information regarding the shape and use of data structures is collected by determining how pointers and heap allocated structures are used. Each piece of information collected is a summary of all pointer paths into and through allocated storage that can arise by executing any path to a particular statement for each statement in the program. [13] provides "methods for determining the class of shapes which an unbounded data object may assume during execution of a LISP-like program". [17] provides a parametric framework for shape analysis, a language for specifying the data properties to be tracked and how these properties change due to the execution of program statements, along with how a shape analysis algorithm results from this language specification. Another approach by [24] derives the forward control dependence relation graph of the program to determine the average-case execution time of the program and its variance. These static techniques differ from *MOQA* not only in their approach but also in their aims. Their focus is to accurately measure certain program attributes, whether it is an evaluation of the program's use of its data structures, the approximate shape of its data structures, or a good estimate of the program's average-case time. The focus of *MOQA* is to exactly measure certain program attributes, to calculate the exact average-case time of a program by knowing the exact shape of its data structures and knowing precisely how they are used. So instead of limiting the information collected about the program, the program itself is limited so that all information required can be collected, with no room for conjecture.

This concept of limiting program functionality to obtain the exact average-case time of a program is a method used by others. This is acknowledged in [8] when discussing the average-case analysis of algorithms, "A path often taken in literature consists in decomposing the structures to be enumerated into smaller structures either of the same type or of simpler types, and then in extracting from such a decomposition recurrence relations...". In [6] a more direct solution based on this approach is presented in the form of LUO, a system that can automatically obtain the non-trivial average-case time of algorithms over a variety of decomposable structures. Some of these non-trivial analyses are given in [4]. LUO utilises the concept of operations, or "admissible constructions", on data structures whose costs can be statically determined, in their case through generating functions as opposed to randomness-preservation in *MOQA*. Randomness-preservation was conceived by Knuth and [15] outlines circumstances where the deletion of an element preserves randomness, under various definitions of the word "random". Random bag preservation is introduced in [18,21,19] as *MOQA*'s novel approach to randomness-preservation.

3 An Overview of *MOQA*

MOQA, like other languages, can be discussed in terms of its data structures and the available primitive language operations on these data structures. Any *MOQA* data structure is a labeled partial order. Every partial order in *MOQA* must begin with its relation being discrete. Any partial order in *MOQA* that has a relation other than discrete must have had that relation constructed through *MOQA* operations. The two *MOQA* functions that enable these other relations to be specified are tightly defined, including `product` and `split`, and can only build a strict subset of all possible relations over a partial order. The fundamental *MOQA* structure is therefore a more complex and tightly constrained representation than an array or list, some basic data structures found in other languages. This fact along with the necessity that any manipulation of a *MOQA* partial order must preserve a specific distribution of the original inputs, i.e. randomness-preservation, also results in primitive language operations upon the series-parallel partial order which are more complex and constrained than primitive language operations are generally found to be. Some of these primitive functions implemented in *MOQA-JAVA* are described informally below and are described formally in work discussing *MOQA* theory [18,21,20]. A high-level description of the `product` operation below is presented in [4,6,8,7]. [6] refers the reader to [9,10,5,11] for earlier discussions of the operation. *MOQA* provides specific details regarding the `product` computation that is randomness-preserving. The *MOQA* split operation below refines both the familiar `partition` operation of Quicksort and its application in [16] to a single labeling of a data structure; it is shown in [18,21,20] to be randomness-preserving over all possible labelings of the data structure.

When the term *component* is used in these descriptions and later on in this paper it refers to a non-empty connected subset of the subset or partial order in

question. The following functions are not the limit of the primitive functions currently available.

Product takes two distinct components of some subset in a partial order and connects one of these components above the other. Prior to `product` the set of elements directly above the two distinct components must be equal. Likewise the set of elements directly below the two distinct components must be equal. After connecting the two components, any values that are now out of order are pushed upwards or downwards until order has been restored.

Relative Delete removes a relative value from a subset in a partial order, e.g. the third biggest value in the subset. The subset must have the property that everything connected above and below the subset is equal to the subtraction of the subset from the component it is contained within. The value to be removed must be relocated to a maximal or minimal element within the subset and its location in one of these elements is ensured by pushing it upwards or downwards until it is correctly located for removal.

Split takes a discrete value in a subset in a partial order and connects all discrete values within the subset that are larger than the specified value above it and connects all discrete values within the subset that are smaller than the specified value below it. Prior to `split` the set of elements directly above each element involved in `split` must be equal. Likewise the set of elements directly below each element involved in `split` must be equal.

These *MOQA* functions, along with others not specified, are not primitive operations in the sense of a standard programming language though they are the most basic functions available in *MOQA*. The *MOQA* language does not provide simpler constructs than these *MOQA* operations, which seems to defy the concept of having simple ideas that can be gathered together to form more complex ideas, as one definition of a powerful programming language requires [1]. With the range of expressiveness that *MOQA* displays it can not be seen as a general-purpose programming language with all the corresponding capabilities of a commonly used language. The compound nature of *MOQA* functions is a necessity for randomness-preservation, remembering that randomness-preservation ensures the average-case time of any *MOQA* program is automatically determined. So the purpose of *MOQA* is not to provide yet another general-purpose programming language but a suite of statically analysable functions. In this context, *MOQA* is no more complex than existing tools that enable the automatic derivation of the average-case complexity of algorithms and in comparison offers a new degree of expressiveness [18]. Regardless of the terminology, whether *MOQA* is seen as a special-purpose programming language, or as a package of functions and rules that determine the behaviour of algorithms that adhere to these rules, the *MOQA* framework addresses an important niche.

4 The *MOQA* Implementation

Currently the language is in the form of a Java 5.0 package and has been named *MOQA-JAVA*, thereby presenting a new paradigm in a familiar setting. However, *MOQA-JAVA* should not be seen as merely an extension to Java, quite the opposite in fact, as *MOQA-JAVA* forbids or restricts some of the basic constructs in the core Java library, such as limiting the range of conditionals allowed in an `if`, `for` and `while` expression and requiring a bound on the number of cycles of a `while` loop. This latter restriction is a practise common in real-time programming for similar timing reasons and therefore just redefines the `for` expression in the syntax of a `while` expression. Also the legal use of *MOQA-JAVA* in conjunction with classes in other Java packages is limited. Therefore, *MOQA-JAVA* can be seen as a language in the Java syntax, which has the object-oriented design of Java but cannot be integrated freely within existing Java legacy code. All code requiring its average-case analysis must hold to the requirements of *MOQA-JAVA*, which differ noticeably from those of standard Java.

The partial order in *MOQA-JAVA* is the `LPO`(Labeled Partial Order) class and the subsets created and returned by the *MOQA* functions are instances of the `SubLPO` class. The `LPO` class is described by the `OrderedCollectionSet` interface and the `SubLPO` class is described by the `OrderedCollectionSubset` interface. Both these interfaces are described by the `OrderedCollection` interface. Each value, which can also be described as a label, and any data associated with it, is stored in an instance of the `NodeInfo` class, which can then be added to one or more `LPO`s. When a `NodeInfo` is added to a `LPO`, the `LPO` creates a `Node` object to contain this `NodeInfo`. A `Node` has a one-to-many relation with a `LPO` and has package-level visibility so any algorithm outside the *MOQA-JAVA* package is unaware of the existence of the `Node` class. The purpose of the `Node` class is to record the relation between a `Node` and the other `OrderedComponent`s within the `OrderedCollection` that directly contains the `Node`. Note that both the `Node` class and the `OrderedCollectionSubset` class inherit the `OrderedComponent` interface. The `OrderedComponent` interface describes the type of object that can be in an `OrderedCollection`. A `Node` is not bound to the `NodeInfo` it initially contains. Due to the nature of some of the *MOQA* functions, such as `product` and `relative delete`, `NodeInfo`s may be swapped between `Node`s but the relation between `NodeInfo`s at any moment in time is recorded by the `Node`s that contain them.

When looking at some algorithms written in *MOQA-JAVA* in the next section it can be seen that the addition of generics to Java 5.0 is used in *MOQA-JAVA* for specifying the value/label type in a `LPO` and all its `SubLPO`s, in other words in an `OrderedCollectionSet` and all its `OrderedCollectionSubset`s.

5 Sorting Algorithms in *MOQA-JAVA*

5.1 Analysis Assumptions

Often when analysing the behaviour of an algorithm the assumption is made that all permutations of the distinct input values are equally likely. Likewise, this assumption is made for all sorting algorithms currently implemented in *MOQA-JAVA*. Knowing the distribution on the inputs can help us to analyse the average-case behaviour of an algorithm, without this knowledge average-case analysis may not be possible in many situations [3]. The analysis assumption that input values are distinct is "fundamental to the analysis of nearly all sorting programs, and it is very often realistic" [22]. However, for *MOQA* this is not the primary motivation behind the assumption of distinct input values but rather that randomness-preservation may be lost if input values are not distinct. Why should the introduction of equal keys cause the loss of randomness-preservation? The problem is that the average-case time of an algorithm with equal keys in the input may not be the same as the average-case time of the same algorithm with distinct keys, which fits in with the results of [22], which uncovered in the case of quicksort that some versions of this algorithm could be quadratic on average when equal keys are involved, a change from quicksort being log-linear on average over distinct keys. So randomness-preservation is a tool for determining the average-case time of a *MOQA-JAVA* algorithm over distinct input values. This average-case time may also hold when duplicate values are included in the input set but there is no promise that it will hold for all input multisets with multiplicity greater than one. Future work will look more closely at the behaviour of algorithms over non-distinct input sets. For the meantime, the distribution on the inputs being equally likely is assumed and the inputs being distinct is mandatory.

The detailed analysis of quicksort presented below is based on the assumption that the following costs take a fixed time:

- the initialisation of a variable,
- the assignment of a value to a variable,
- each arithmetic operator,
- a boolean comparison,
- accessing an item in an array by its index,
- the *instanceof* keyword,
- the *new* keyword. This final fixed time cost does not cover the cost of the operations within the constructor of the newly created object. If any operations are present in the constructor their cost is also calculated.

A further assumption is made regarding these fixed costs, that they are all equal. This assumption can be replaced by a more refined estimation at a later time.

```
/**
 * Sorts the specified OrderedCollection according to the
 * natural ordering of its NodeInfos.
 * @param oc the OrderedCollection to be sorted.
 */
public static <L extends Comparable<L>> void
        insertionsort(OrderedCollection<L> oc)
{   if (oc.size() > 1)
    {   Iterator<NodeInfo<L>> ocNodeInfos =
                oc.getDirectNodeInfoIter();
        OrderedCollectionSubset<L> sorted =
                oc.product(ocNodeInfos.next(),
                        ocNodeInfos.next());
        while (ocNodeInfos.hasNext())
        {   sorted = oc.product(ocNodeInfos.next(), sorted);
        }
    }
}
```

Fig. 1. Insertion-sort in *MOQA-JAVA*

5.2 Sorting Algorithm Examples

5.2.1 Insertion-sort

The pseudo-code for this well-known algorithm, [3], commonly used for its efficiency in sorting small data sets is as follows:

Insertion-Sort(A)
for $j \leftarrow 2$ to $length[A]$
  do $key \leftarrow A[j]$
    ▷ Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$
    $i \leftarrow j-1$
    while $i > 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i-1$
    $A[i+1] \leftarrow key$

Figure 1 shows insertion-sort implemented in *MOQA-JAVA*.

After comparing the insertion-sort pseudo-code to the *MOQA-JAVA* implementation of insertion-sort, it is clear that *MOQA* provides another level of abstraction. With *MOQA-JAVA* there is no explicit reference to the position of the next element to be inserted correctly amongst the elements already sorted, whereas in the pseudo-code the explicit reference to this position is the variable j, the index in the array of the next item to be inserted. In *MOQA-JAVA* an Iterator over the `OrderedCollection` to be sorted returns the next element for insertion. The first two elements returned by the It-

erator are the parameters for the first `product`. `Product` removes these elements from the specified `OrderedCollection` *oc*, the `OrderedCollection` to be sorted, and connects the greater of the two above the lesser within a *new* `OrderedCollectionSubset` that is added to *oc* before being returned. After this for each `product`, a `NodeInfo` in *oc* is removed from *oc* and connected above the specified `OrderedCollectionSubset` to form a new `OrderedCollectionSubset` that replaces the one previously added to *oc*. After being connected above the previously returned `OrderedCollectionSubset`, the `NodeInfo` is pushed into its correct position in this newly created `OrderedCollectionSubset`, which is then returned by `product`.

From this we can see that a *MOQA-JAVA* Iterator is *not fail-fast*: if the `OrderedCollection` is modified at any time after the Iterator over it is created the Iterator does not fail. This is because the Iterator is actually created over a copy of the `OrderedCollection`'s content. Any iteration, whether partial or complete, over an `OrderedCollection` must always return the `NodeInfo`s in the order that the `Node`s they are attached to are stored in the `OrderedCollection`. The `Node`s are stored in the order that they were added to the `OrderedCollection`. This ordering over the `Node`s as opposed to the `NodeInfo`s is necessary for randomness-preservation.

It is also clear that the in-place nature of insertion sort is lost in *MOQA-JAVA* as a new `OrderedCollectionSubset` is created for every element in the `OrderedCollection` to be sorted, excluding the first two elements for which a single `OrderedCollectionSubset` is created. So the actual parameters for each `product` are removed from the specified `OrderedCollection` *oc* and added to a newly created `OrderedCollectionSubset` that is in turn added to *oc*. This results in the creation of $n-2$ `OrderedCollectionSubset`s so $n-2$ extra space is required in total: $n-3$ for the number of `OrderedCollectionSubset`s that contain one element and the additional reference to an `OrderedCollectionSubset` plus 1 for *oc* that also contains one element and the additional reference to an `OrderedCollectionSubset`. Actually this $n-2$ cost applies if *oc* is an instance of `OrderedCollectionSubset`. Otherwise if it is an `OrderedCollectionSet` an extra $n-1$ space, as opposed to $n-2$, is required due to the fact that an `OrderedCollectionSet` only contains components as direct content.

So the requirement that the *MOQA* data structure is a partial order results in its traversal and manipulation being more intricate than that of an array. The current implementation is one way of storing this additional complexity. So for this implementation, what is the price of this extra information in the average-case time of an algorithm? It may be more useful to examine this question with a sorting algorithm that is not so simplistic in its approach to sorting.

5.2.2 Quicksort

The next algorithm presented is quicksort, one of the more interesting algorithms in terms of the difference between the asymptotic timing of its average and worst case. The pseudo-code for quicksort, [3], to sort an array $A[p \dots r]$ is:

```
/**
 * Sorts the specified OrderedCollection according to the
 * natural ordering of its NodeInfos.
 * @param oc the OrderedCollection to be sorted.
 */
public static <L extends Comparable<L>> void
        quicksort(OrderedCollection<L> oc)
{   NodeInfo<L> partitionNI =
            oc.getDirectNodeInfoIter().next();
    OrderedCollection<L> partition = oc.split(partitionNI);
    Iterator<OrderedCollectionSubset<L>> aboveAndBelow =
            partition.getDirectSubsetIter();
    if (aboveAndBelow.hasNext())
    {   quicksort(aboveAndBelow.next());
        if (aboveAndBelow.hasNext())
        {   quicksort(aboveAndBelow.next());
        }
    }
}
```

Fig. 2. Quicksort in *MOQA-JAVA*

Quicksort(A, p, r)
if $p < r$
  then $q \leftarrow$ *Partition*(A, p, r)
    Quicksort$(A, p, q-1)$
    Quicksort$(A, q+1, r)$

Partition(A, p, r)
$x \leftarrow A[r]$
$i \leftarrow p-1$
for $j \leftarrow p$ to $r-1$
  do if $A[j] \leq x$
    then $i \leftarrow i+1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
return $i+1$

Figure 2 shows quicksort implemented in *MOQA-JAVA*.

The most basic recurrence for this standard quicksort that does not take advantage of the optimisation techniques presented by [14,12,23] is:

$$T(n) = n - 1 + \frac{2}{n}\sum_{k=1}^{n} T(k-1) \tag{1}$$

It has been pointed out that *MOQA* functions are not as primitive as those found in general programming languages, in *MOQA-JAVA* they are composed of many

Java primitive operations. For the case of quicksort the *MOQA* function involved is as complex as the algorithm itself. Clearly `split` is in essence `partition`, the key element of quicksort, and the consequence of `split` on a partial order is that it contains a partial order above the pivot, if there are elements larger than the pivot, and a partial order below the pivot, if there are elements smaller than the pivot. So how is the average-case time of quicksort affected by a data structure whose content relation is predefined by the language as opposed to a data structure whose content relation is determined by the algorithm as in standard quicksort?

The average-case recurrence for *MOQA-JAVA* quicksort on a discrete partial order is:

$$quicksort(n) = c_1 n + split(n) + c_2\left(\frac{3\,(n-4)+10}{n}\right) + c_3\left(\frac{2\,(n-4)+4}{n}\right) + \\ + c_4 + \frac{2}{n}\sum_{k=1}^{n} quicksort(k-1), \qquad n > 3 \tag{2}$$

with the first term $c_1 n$ representing the cost of getting an Iterator over the `NodeInfo`s in the specified `OrderedCollection` *oc*, the third term $c_2\left(\frac{3(n-4)+10}{n}\right)$ representing the cost of getting an Iterator over the `OrderedCollectionSubset`s in the specified `OrderedCollection` *oc* after the `split` operation and the fourth term $c_3\left(\frac{2(n-4)+4}{n}\right)$ representing the cost of making the recursive calls. c_1, c_2 and c_3 are constants in these costs and c_4 represents the other constant costs that occur in a call to quicksort.

The recurrence for *MOQA-JAVA* quicksort is to be obtained by a separate program statically analysing the code in Figure 2. As this static analysis program is still under construction, recurrence 2 and the rest of the recurrences presented here were obtained by a careful hand analysis of Figure 2 and *MOQA-JAVA* functions. However, the static analysis program should produce similar recurrences. How the static analysis tool calculates such recurrences automatically is addressed in [18]. As it is the recurrences themselves that can answer the question above, the following conclusions will not change, regardless of whether they are calculated manually or automatically.

Recurrence 2 can be simplified to:

$$quicksort(n) = c_1 n + split(n) + 3c_2 + 2c_3 - \frac{2}{n}(c_2 + 2c_3) + c_4 + \\ + \frac{2}{n}\sum_{k=1}^{n} quicksort(k-1), \qquad n > 3 \tag{3}$$

The second term in the recurrence is $split(n)$. `Split` is over a partial order of size n, where a partial order is a pair $(X, \sqsubseteq)$ consisting of a set X and a binary relation $\sqsubseteq$ between elements of X such that the relation is reflexive, transitive and anti-symmetric. For `split` in *MOQA-JAVA*'s quicksort, X is the specified `OrderedCollection` *oc* and the order $\sqsubseteq$ on X is restricted to the discrete order. The elements of X are stored in the collection C_X within the `OrderedCollection` object that represents X. The pivot element is the single element p that is a

component in X. `Split` can be broken down into this sequence of events:

(i) $Y = getDiscrete(X)$
Records in the set Y the discrete elements present in X.

(ii) $removePivot(Y, p)$
Remove p from Y.

(iii) $A, B = relation(Y, p)$
Records in the set A the elements in Y that are greater than p and records in the set B the elements in Y that are smaller than p.

(iv) $relocate(A, B)$
If $|A| > 1$, then A is removed from C_X and stored in the collection C_A within a new `OrderedCollectionSubset` object, which is added to C_X. If $|B| > 1$, then B is removed from C_X and stored in the collection C_B within a new `OrderedCollectionSubset` object, which is added to C_X.

(v) $connect(A, p, B)$
If $|A| > 0$ it is connected above p and if $|B| > 0$ it is connected below p. The elements of X remain the same but the binary relation between them has now changed.

The pivot element p is removed from Y before determining which elements in Y are greater than p and which elements in Y are smaller than p. The reason for this was to reduce the cost of $relation(Y)$. If p was not removed from Y it would be necessary, prior to checking whether each element in Y is larger than p, to ensure that the element was not itself p. Leaving aside other costs, this would give $2n - 1$ comparisons instead of $n - 1$ as seen in the standard recurrence, a difference of n on average. But what about the cost of $removePivot(Y, p)$? In the *MOQA-JAVA* quicksort code above, the pivot element p is always the `NodeInfo` of the first `Node` stored in C_X, as the first `NodeInfo` returned by the Iterator is specified as the pivot element. Therefore the cost of removing the pivot element p is the cost of removing the first `Node` in C_X, which is constant on average.

More succinctly, `split` in *MOQA-JAVA*'s quicksort is a series of:

$$\begin{aligned}\overline{T}(qsSplit(X,p)) = {} & \overline{T}(getDiscrete(X)) + \overline{T}(relation(Y)) + \\ & + \overline{T}(relocate(A,B)) + \overline{T}(connect(A,p,B)) + c_{qsSplit}\end{aligned} \tag{4}$$

When these functions are replaced by their average-case time, average-case time being analysed using the assumptions in 5.1, the equation becomes:

$$\begin{aligned}qsSplit(n) = {} & (23n) + \left(17\,(n-1) + 6n + \frac{23\,(n-1))}{n}\right) + \\ & + \left(\frac{2\,(40n + c_r + c_a + 19)}{n} + \frac{2\,(40n + c_r + c_a + 26)}{n} + \right. \\ & \left. + 5\left(\frac{n+2}{n}\right) + \frac{\sum_{k=3}^{n-2} 32\,(k-1) + 24n + c_r + 2c_a}{n}\right) + \\ & + \left(\frac{4n-4}{n}\right) + c_{qsSplit}, \qquad n > 3\end{aligned} \tag{5}$$

where c_r is the constant for the function called within $relocate(A, B)$ that removes the specified set from C_X and c_a is the constant for the function called within $relocate(A, B)$ that adds the specified set to a new `OrderedCollectionSubset`. This equation is reduced to:

$$(6)\quad qsSplit(n) = 100n + \frac{26}{n} - \frac{4c_a}{n} - \frac{9}{2} + c_{split}, \qquad n > 3$$

For insertion-sort, the extra space cost differed slightly depending on whether the specified `OrderedCollection` *oc* was an instance of `OrderedCollectionSet` or `OrderedCollectionSubset`. The same is true for `split`, its cost varies slightly depending on whether the specified `OrderedCollection` *oc* is an instance of `OrderedCollectionSet` or `OrderedCollectionSubset`. This difference in cost is again tied to the design decision that an `OrderedCollectionSet` only contains components as direct content. So equation 6 is the common average-case cost of `split` regardless of the type of the specified `OrderedCollection` *oc*. The total equation for `split` to be substituted for $split(n)$ in recurrence 3 when the specified `OrderedCollection` *oc* is an `OrderedCollectionSet` is:

$$(7)\quad splitOnSet(n) = qsSplit(n) + \frac{64}{n} + 64 + c_{splitOnSet}, \qquad n > 3$$

The total equation for `split` to be substituted for $split(n)$ in recurrence 3 when the specified `OrderedCollection` *oc* is an `OrderedCollectionSubset` is:

$$(8)\quad splitOnSubSet(n) = qsSplit(n) + c_{splitOnSubSet}, \qquad n > 3$$

It will always be $splitOnSubset(n)$ that is substituted for $split(n)$ in recurrence 3 if only part of the overall `OrderedCollectionSet` is being sorted. If the entire `OrderedCollectionSet` is to be sorted, $splitOnSet(n)$ is substituted once for the first call to $quicksort(n)$ and $splitOnSubset(n)$ is substituted from then onwards.

The initial effort to reduce the cost of $relation(Y)$ from $2n - 1$ to $n - 1$ gave a saving of n. This saving clearly got swamped by the overall cost of `split` for *MOQA-JAVA*'s quicksort. More tweaking of split in *MOQA-JAVA* will no doubt further reduce these constant values but *MOQA* functions provide another level of abstraction between the programmer and the data structure, itself more complex than what is normally used by sorting algorithms under analysis. Combined with the fact that *MOQA* functions are designed to be general-purpose and are not geared towards one specific algorithm, means that it is unlikely that quicksort's constants will ever be reduced to the constants in Sedgewick's non-optimised version of quicksort [23]. So while *MOQA-JAVA*'s quicksort does not affect the asymptotic average of quicksort, $\Theta(nlogn)$, it does increase the constant values.

6 Summary

MOQA-JAVA is the current implementation of the *MOQA* approach. In providing randomness-preservation for the static analysis of algorithms, *MOQA* functions remove certain actions normally carried out directly within an algorithm and execute these actions internally. As the *MOQA* functions have to be generic for use in a wide range of permissible situations, the challenge is to implement the *MOQA*

functions as efficiently as possible without a loss of this generality. The two sorting algorithms presented in this paper elaborate on some of the costs that the *MOQA* functions in the current implementation generate.

Insertion-sort in *MOQA-JAVA* shows an additional cost in space and from examining the average-case behaviour of quicksort in *MOQA-JAVA* we can see a tendency towards higher constant values than normal, though the recurrence does not deviate from the standard trend of quicksort. To achieve static analysis it is not deemed unacceptable to carry some extra expense when it does not change the asymptotic behaviour of the algorithm. While it is the aim to lower these costs to their minimum value, it can be expected, in order to statically determine the average-case time of an algorithm, that the constants will be higher within the same order of growth as traditional variants.

7 Future Work

Future work includes:

- Further effort to optimise *MOQA* functions in the current implementation.
- Refactoring *MOQA-JAVA*. While the encapsulation of the partial order data structure within the `OrderedCollection` class is important, the indirect manipulation of it through the current API is not flexible enough. This and other refactoring will take place, along with an extension to *MOQA-JAVA* that will include more specific partial orders types.
- To date a narrow range of algorithms are expressed in *MOQA*, these being sorting algorithms and quickselect, a search algorithm based upon quicksort. Work is ongoing to extend this range of algorithms.
- Exploring the average-case analysis of algorithms when there is duplication in the input.

References

[1] H. Abelson, G.J. Sussman, and J.Sussman. *Structure and Interpretation of Computer Programs, 2/e.* MIT Press, 1996.

[2] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *PLDI*, 1990.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, 2/e.* MIT Press and McGraw-Hill, 2001.

[4] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-upsilon-omega: The 1989 cookbook. *Research Report 1073, Institut National de Recherche en Informatique et en Automatique*, 1989.

[5] Philippe Flajolet. Analyse d'algorithmes de manipulation d'arbres et de fichiers. *Cahiers du Bureau Universitaire de Recherche Oprationnelle, Vol 34-35*, 1981.

[6] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 1991.

[7] Philippe Flajolet and Robert Sedgewick. Analytic combinatorics. *Chapters I-IX of a book to be published by Cambridge University Press, available electronically from P. Flajolet's home page.*

[8] Philippe Flajolet and Robert Sedgewick. The average case analysis of algorithms: Counting and generating functions. *Research Report 1888, Institut National de Recherche en Informatique et en Automatique*, 1993.

[9] Dominique Foata. *La srie gnratrice exponentielle dans les problmes d'numration.* University of Montral Press, 1974.

[10] I. Goulden and D. Jackson. *Combinatorial Enumerations.* Wiley, New York, 1983.

[11] D. H. Greene. Labelled formal languages and their uses. *Stanford University, Technical Report STAN-CS-83-982*, 1983.

[12] C.A.R. Hoare. Quicksort. *Computer Journal Vol 5, Number 1*, 1962.

[13] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. *POPL*, 1979.

[14] D.E. Knuth. *The Art of Computer Programming, Volume 3.* Addison-Wesley, 1973.

[15] D.E. Knuth. Deletions that preserve randomness. *IEEE Transactions on Software Engineering, Vol SE-3, No. 5*, 1977.

[16] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[17] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *POPL*, 1999.

[18] M. Schellekens. *A Modular Calculus for the Average Cost of Data Structuring.* To appear with Springer Verlag, 2007.

[19] M. Schellekens. Moqa; unlocking the potential of compositional static average-case analysis. *Internal CEOL Report*, 2007.

[20] M. Schellekens. A randomness preserving product operation. *Electronic Notes in Theoretical Computer Science*, 2007.

[21] M. Schellekens. Sequential compositionality: Removing a fundamental bottleneck from average-case analysis. *Internal CEOL Report*, 2007.

[22] R. Sedgwick. Quicksort with equal keys. *SIAM Journal on Computing Vol 6, Number 2*, 1977.

[23] R. Sedgwick. Implementing quicksort programs. *Comm. ACM Vol 21, Number 10*, 1978.

[24] V.Sarkar. Determining average program execution times and their variance. *PLDI*, 1989.