Classifying and Recognizing Students' Sorting Algorithm Implementations in a Data Structures and Algorithms Course

Ahmad Taherkhani, Ari Korhonen, Lauri Malmi

Aalto University

SCIENCE + TECHNOLOGY

RESEARCH REPORT

Aalto University publication series
SCIENCE + TECHNOLOGY 4/2012

Classifying and Recognizing Students' Sorting Algorithm Implementations in a Data Structures and Algorithms Course

Ahmad Taherkhani, Ari Korhonen, Lauri Malmi

Aalto University
School of Science
Department of Computer Science and Engineering
Learning + Technology Group, LeTech

Aalto University publication series
SCIENCE + **TECHNOLOGY** 4/2012

ISBN 978-952-60-4526-9 (pdf)
ISSN-L 1799-4896
ISSN 1799-490X (pdf)

Unigrafia Oy
Helsinki 2012

Finland

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author

Author(s): Ahmad Taherkhani, Ari Korhonen, Lauri Malmi

Name of the publication

Classifying and Recognizing Students' Sorting Algorithm Implementations in a Data Structures and Algorithms Course

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series SCIENCE + TECHNOLOGY 4/2012

Field of research Software Systems

Abstract

We discuss an instrument for recognizing and classifying algorithms (Aari) in terms of white-box testing. We examined freshmen students' sorting algorithm implementations in a data structures and algorithms course in two rounds: at the beginning of the course before the students received any instruction on sorting algorithms, and after taking lectures on sorting algorithms. We evaluated the performance of Aari with the implementations of each round separately. The results show that the sorting algorithms (in Java), which Aari has been trained to recognize (Insertion sort, Selection sort, Bubble sort, Quicksort and Mergesort), are recognized with an average accuracy of about 90%. When considering all the submitted sorting algorithm implementations (including the variations of the standard algorithms), Aari achieved an overall accuracy being 71% and 81% for the first and second round, respectively.

The manual analysis of the implementations revealed that students have many misconceptions related to sorting algorithms. For example, they include unnecessary swaps in their Insertion or Selection sort implementations. Based on the data, we present a categorization of these types of variations. We also discuss how these categories can be used to improve education of sorting methods, and to further develop Aari as a white-box testing tool, which gives feedback to the students on their inappropriate solutions and asks them to correct the problem; something black-box testing is not capable of doing.

Keywords Algorithm recognition, classifying students' sorting implementations, roles of variables, automatic assessment tools, white-box testing

ISBN (printed) **ISBN (pdf)** 978-952-60-4526-9

ISSN-L 1799-4896 **ISSN (printed)** 1799-4896 **ISSN (pdf)** 1799-490X

Location of publisher Espoo **Location of printing** Helsinki **Year** 2012

Pages 21

Classifying and Recognizing Students' Sorting Algorithm Implementations in a Data Structures and Algorithms Course

AHMAD TAHERKHANI, ARI KORHONEN, and LAURI MALMI
Aalto University
Department of Computer Science and Engineering
P.O.Box 15400, FI-00076 AALTO, Finland
{ahmad, archie, lma}@cs.hut.fi

We discuss an instrument for recognizing and classifying algorithms (Aari) in terms of white-box testing. We examined freshmen students' sorting algorithm implementations in a data structures and algorithms course in two rounds: at the beginning of the course before the students received any instruction on sorting algorithms, and after taking lectures on sorting algorithms. We evaluated the performance of Aari with the implementations of each round separately. The results show that the sorting algorithms (in Java), which Aari has been trained to recognize (Insertion sort, Selection sort, Bubble sort, Quicksort and Mergesort), are recognized with an average accuracy of about 90%. When considering all the submitted sorting algorithm implementations (including the variations of the standard algorithms), Aari achieved an overall accuracy being 71% and 81% for the first and second round, respectively.

The manual analysis of the implementations revealed that students have many misconceptions related to sorting algorithms. For example, they include unnecessary swaps in their Insertion or Selection sort implementations. Based on the data, we present a categorization of these types of variations. We also discuss how these categories can be used to improve education of sorting methods, and to further develop Aari as a white-box testing tool, which gives feedback to the students on their inappropriate solutions and asks them to correct the problem; something black-box testing is not capable of doing.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

General Terms: Algorithms, Design, Human factor

Additional Key Words and Phrases: Algorithm recognition, classifying students' sorting implementations, roles of variables, automatic assessment tools, white-box testing

1. INTRODUCTION

Many automatic assessment systems are based on black-box testing, that is, with certain input the output is tested by comparing it with an output produced by a model solution. This approach cannot say anything about the quality of the solution algorithm. For example, novice programmers can often implement a working algorithm that passes the black-box testing, but includes problematic implementation choices that make the solution impractical or inefficient. We will demonstrate these in the case of sorting algorithms. In addition, black-box testing cannot force using a certain algorithm.

White-box testing is supposed to say something about the internal structure of an algorithm. In mixed methods, both black-box and white-box testing are used. We can, for example, first test the functionality in terms of black-box testing. If the solution works, then white-box testing techniques can be applied to give feedback about the problematic design choices. In the following, we address only the white-box testing part assuming that the functionality of the algorithm is already tested in terms of black-box testing.

In this paper, we discuss and evaluate a prototype instrument to inspect students' programming exercises in terms of white-box testing. The instrument is based on static analysis of program code, and its current version can classify student submissions for a sorting algorithm exercise. Our aim is to develop this instrument further so that it could be used as a part of automatic assessment system to give meaningful feedback for both students and teachers on a wide variety of programming tasks involving implementing algorithms. For the rest of the paper, we call this instrument *Aari* (an Automatic Algorithm Recognition Instrument).

In order to test the new tool we collected freshmen students' submissions in a task where they were requested to write a simple program to sort an array of integers. The submitted programs were evaluated both manually and with Aari, which allowed us to compare the results and evaluate the performance of Aari. Repeating the assignment later, after the course lectures had covered sorting algorithms, also enabled us to compare the changes in students' implementations.

Our research questions are:

(1) *How can we classify students' implementations of sorting algorithms and what kind of variations of well-known sorting algorithms they use?*
(2) *How well can Aari recognize five basic sorting algorithms that it has been trained to recognize (Insertion sort, Selection sort, Bubble sort, Quicksort and Mergesort) and their variations among the students' submissions?*

The manual analysis revealed that students have many misconceptions related to sorting algorithms. We believe that the same also holds for other topics in basic programming courses, especially related to data structures and algorithms (see, for example, [Seppälä et al. 2006]). One of our future objectives concerning Aari is to detect such misconceptions automatically so that the system could give feedback on the problem and ask the student to correct the solution. For example, when implementing Insertion or Selection sort, many students include unnecessary swaps in their code. This makes the code more inefficient and more complicated to understand and maintain. Existing automatic assessment systems are based on black-box testing and they are not capable of detecting such problems due to the fact that the code still works correctly and sorts the given input as specified. Based on this and similar observations which emerged from the analysis of the student implementations and will be presented in this paper, we will further develop Aari as a white-box testing tool to classify such incorrect implementations and separate them from correct implementations. This classification can be used to give feedback to the students on inappropriate solutions.

It is notable that even some of the textbooks introduce the aforementioned problematic versions of the algorithms (e.g., Insertion sort with swapping elements instead of shifting the elements). We would like to provide feedback that could make the student think his solution anew. Blindly copying an algorithm from a textbook or a web resource is not necessarily the best option. Many textbooks introduce only the very basic version of an algorithm. This can lead to serious performance problems. For example, Quicksort requires very well-aimed modifications, and the typical basic version in textbooks have its well-known $O(n^2)$ worst-case performance with already sorted input. These are just examples of problems that might occur in students' solutions and that need to be addressed in all data structures and algorithms courses. Automatic feedback in case of large courses is essential to make sure that students learn this also in practice.

Aari has its limitations. One of the consequences of the Rice's theorem (and undecidability of the halting problem) is the fact that there cannot be a general algorithm that decides whether two algorithms compute the same function. Thus, even though we had a "model solution" for all the possible sorting algorithms, we cannot always (in general) decide whether the student's implementation corresponds to the same method or not. However, in our case, infinite exactness of the decision is not required. Even humans make errors, and cannot give feedback with 100% accuracy. Thus, having an instrument that can classify the solutions in a *reasonable* precision is sufficient. Thus, one of the objectives in this paper is to measure the accuracy of Aari.

The rest of the paper is divided into the following sections. We briefly discuss related work in Section 2. We present Aari in Section 3, where we also discuss the recognition process and the code characteristics used in the recognition process. In Section 4 we describe the data structures and algorithms course and the data collection and preparation process. Section 5 presents the categories of sorting algorithms emerged from the student's implementations. Different algorithm implementations submitted by the students as well as the results of evaluating Aari with these implementations are presented in Section 6. We discuss the results in more detail in Section 7. Finally, Section 8 presents some conclusions and explains the directions for future work.

2. RELATED WORK

A number of automatic assessment systems have been introduced for assessing different types of programming exercises. We first present some of the functionalities these systems have. Then we discuss a study conducted on what students know about sorting algorithms before they have been formally instructed.

Automatic assessment tools

Automatic assessment systems are widely used by instructors for grading student works in large programming and data structure courses. They reduce the workload of instructors and support learning by giving quick feedback to students.

These systems have several different functionalities varying from checking the correctness and functionality of programs to analyzing program structure (see, for example, WebCat [Edwards 2003], CourseMarker [Higgins et al. 2002] and BOSS [Joy et al. 2005]).

Some systems can analyze and check how well the program has been tested [Edwards 2003], the programming style [Elenbogen and Seliya 2007], the use of various language constructs [Saikkonen et al. 2001], memory management [Ala-Mutka and Järvinen 2004], and run time efficiency [Saikkonen et al. 2001].

An overview of the field and information about these functionalities can be found in [Ala-Mutka 2005].

The existing automatic assessment systems, however, cannot perform white-box testing in the sense of analyzing how the given problem has been solved, that is, they are not able to check whether students have implemented the required algorithm, or the quality of the solution. Aari has been developed to address this deficiency.

Students' knowledge of sorting algorithms

A series of studies have been conducted to investigate what students already know before they have been formally taught by instructors. One of these studies is about students' knowledge of sorting algorithms [Simon et al. 2006], where Simon et al. asked the students on the first day of a CS1 course, a sub-population of this CS1 students after ten week of instruction, and the students of an introductory economic course, to describe in English how they would sort a set of 10 numbers in ascending order. They found that a majority of computer science students provided a logical description to the problem, while only less than a third of the other students did so. They divided the described algorithms into six categories. They found that most of the students described algorithms that processed the numbers as strings of digits and dealt with them digit by digit. The second most common descriptions were similar to Selection sort. Some of the provided descriptions were like Insertion sort and a few students gave Bubble sort algorithms as the answer. Furthermore, three students described an algorithm which Simon et al. call the "pessimal algorithms" (select the number closest to 1 as the first number and keep adding 1 until the next number is reached and so on) and three students gave an algorithm which the authors call "the number line algorithm" (place each number on its correct location on the number line and read them from left to right). We will get back to this in Section 7 where we compare our findings with those of Simon et al.

3. AARI

We have previously developed Aari for recognizing algorithms. Aari and the recognition method is described in [Taherkhani et al. 2011] and [Taherkhani 2011]. In this section, we briefly outline the recognition method. The reader is encouraged to read our aforementioned previous articles for more information.

The method we have used in algorithm recognition is based on static analysis of source code. Several characteristics of source code are analyzed, including various statistics of language constructs, software metrics as well as roles of variables appearing in the target program code. Once these characteristics are computed, they are stored in a database and are used for building a decision tree that guides the recognition process. The C4.5 decision tree classifier algorithm is used to build the tree by identifying those characteristics that can best distinguish between the analyzed algorithms.

Computed characteristics

The characteristics computed for each algorithms are shown in Table I. The characteristics are divided into *numeric characteristics*, *descriptive characteristics*, and *other characteristics*. The numeric characteristics are those that can be measured as integers, whereas the descriptive characteristics reveal those kind of features of implementations of algorithms that cannot be measured as integers. In order to be used in the analysis, after being computed, the descriptive characteristics are converted into 0 or 1 indicating the existence or absence of the corresponding characteristics. Other characteristics, shown in the third part of Table I are used to computed different patterns that can be used in the recognition process. For example, the last two characteristics in the table, *OIID* (Outer loop Incrementing Inner Decrementing) and *IITO* (Inner loop counter Initialized To Outer loop counter), are computed using the characteristics *Loop counter information*, *Dependency information* and *NoNL* (Number of Nested Loops). These characteristics illustrated at the end of the table are used in the process of recognizing sorting algorithms, which is the topic of the experiment described later in this paper.

Roles of variables. Roles of variables (RoV) [Sajaniemi 2002] are specific patterns how variables are used in source code and how their values are updated during program execution. RoV are tacit programming knowledge [Sajaniemi and Prieto 2005]. Although RoV were originally introduced to help students learn programming, the concept can offer an effective tool to analyze programs with different purposes. As also discussed in our previous work [Taherkhani et al. 2011; Taherkhani 2011], we use RoV in algorithm recognition. RoV can be analyzed automatically using data flow analysis and machine learning techniques (see [Bishop and Johnson 2005] and [Gerdt and Sajaniemi 2006]).

Sajaniemi identified nine roles that cover 99% of all variables used in 109 novice-level procedural programs [Sajaniemi 2002]. Currently, based on a study on applying the roles in object-oriented, procedural and functional programming [Sajaniemi et al. 2006], the following 11 roles are recognized[1]. Note that the three last roles in the list are related to data structures.

(1) *Stepper*: goes through a succession of values.

[1]See the RoV Home Page `http://www.cs.joensuu.fi/~saja/var_roles/` for a more comprehensive information on roles

Table I. The numerical, descriptive and other related characteristics computed from algorithms. The last five descriptive characteristics are specific to the sorting algorithms

Numerical characteristics	**Description**
NAS	Number of assignment statements.
LoC	Lines of code.
MCC	McCabe complexity (i.e., cyclomatic complexity) [McCabe 1976].
N_1	Total number of operators.
N_2	Total number of operands.
n_1	Number of unique operators.
n_2	Number of unique operands.
N	Program length ($N = N_1 + N_2$).
n	Program vocabulary ($n = n_1 + n_2$).
NoV	Number of variables.
NoL	Number of loops. Supported loops are *for* loop, *while* loop and *do while* loop.
NoNL	Number of nested loops.
NoB	Number of blocks.
Descriptive characteristics	**Description**
Recursive	Is the algorithm recursive.
Tail recursive	Is the algorithm tail recursive.
Roles of variables	Roles of the variables used in the implementation of the algorithm.
Auxiliary array	For the implementations of algorithms that use array, is an auxiliary array used?
Other characteristics	**Description**
Block/loop information	Information about blocks and loops, including starting and ending lines, length and interconnection between them (how they are positioned in relation to each other).
Loop counter information	Information about how the value of loop counters are initialized and updated. This is used to determine incrementing/decrementing loops (the value of the loop counter increases/decreases after each iteration).
Dependency information	Direct and indirect dependencies between variables (variable i is directly dependent on variable j, if i gets its value directly from j. If there is a third variable k on which j is directly or indirectly dependent, i also becomes indirectly dependent on k. A variable can be both directly and indirectly dependent on another one).
Characteristics for sorting algorithms	**Description**
MWH	Does the implementation of the algorithm use a most-wanted holder variable.
TEMP	Does the implementation of the algorithm use a temporary variable.
In-place	Does the algorithm need extra memory.
OIID	(Outer Incrementing Inner Decrementing) Whether from the two nested loop used in the algorithm, the outer is an incrementing loop and the inner is a decrementing loop.
IITO	(Inner Initialized To Outer) Whether from the two nested loop used in the algorithm, the inner loop counter is initialized to the value of the outer loop counter.

(2) *Temporary*: holds a value for a short period of time.
(3) *Most-wanted holder*: holds the most desirable value that is found so far.
(4) *Most-recent holder*: holds the latest value from a set that is being gone through, or holds the latest input value.
(5) *Fixed value*: keeps its value throughout the program.
(6) *One-way flag*: can have only two values and once its value is changed, it cannot get its previous value back.
(7) *Follower*: always gets its value from another variable, (i.e., its new values are the old values of other variables).
(8) *Gatherer*: collects the values of other variables.
(9) *Organizer*: stores data elements that can be rearranged.
(10) *Container*: stores data elements to be added or removed.
(11) *Walker*: traverses data structures.

For automatic detection of roles of variables in this and our previous studies, we used a tool developed by Bishop and Johnson [Bishop and Johnson 2005]. We did several improvements to the tool before using it. As an example, for each variable appearing in the target program, the tool requires that special tags along with the name of the vari-

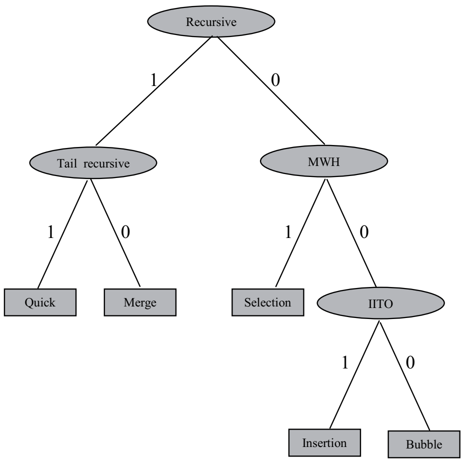

Fig. 1. Decision tree used in the experiment. The tree is built by the C4.5 algorithm and is described in more detail in our previous paper [Taherkhani 2011]

able and some string as the role is provided by the user, otherwise the tool will ignore the variable. We further developed the tool so that the name of all the variables of the program along with the required tags and the required string are provided automatically. We also tuned up the tool in order to improve its performance. See [Taherkhani 2011] for more information about these improvements. For more information on how the role detector works see [Bishop and Johnson 2005].

Recognition process

After computing the aforementioned characteristics and storing them in a database, each algorithm can be presented as a characteristic vector in the database. These characteristic vectors can be analyzed by a suitable decision tree classifier to generate an automatic decision tree that can be used for classifying algorithms. We have previously reported a study in [Taherkhani 2011], where we used 209 Java implementations of five types of sorting algorithms (Selection sort, Insertion sort, Bubble sort, Quicksort and Mergesort) as learning data to build a decision tree that can be used in recognition of these sorting algorithms. The decision tree was built using the C4.5 algorithm. In [Taherkhani 2011], we also presented the results of evaluating the average classification accuracy using leave-one-out cross-validation technique and showed that using the instances of the data set of 209 sorting algorithms, the average accuracy was 98.1%.

For recognizing the student works in the experiment presented in this paper, we applied the same algorithm and used the same data set as presented in [Taherkhani 2011] to build the decision tree of Figure 1. Note that since the algorithms of the data set are used as learning data to build the decision tree, each algorithm in the database is labeled by its correct type.

As can be seen from the figure, the tree has four internal nodes (with the root included). These nodes are illustrated as ellipses and include the tests based on which the splits are performed. In other words, from the computed characteristics discussed above, the values of four characteristics are used for building the decision tree. There are five leaves in the tree, which correspond to the number of different types of the sorting algorithms and are labeled with the appropriate algorithm. From each internal node, there are arcs labeled with 0 or 1 to its children. Each value of an arc illustrates the outcome of the corresponding test performed in the internal node in question. As an example, the arc labeled with 1 goes from the internal node Tail recursive to the left and the other one labeled with 0 goes to the right; the former indicating that tail recursive algorithms belong to the leaf labeled with “Quick”, and the latter indicating that algorithms that are not tail recursive belong to the leaf labeled with “Merge”.

The aforementioned characteristics of the student works are computed and stored in the database. These algorithms constitute the testing data, and thus they are labeled as unknown in the database. Once all the algorithms of the testing data are stored as the characteristic vectors in the database, they are classified and assigned a type by the decision tree classifier one by one.

At present, the decision tree of Figure 1 has a mechanism to recognize only the five aforementioned sorting algorithms. Recognizing other sorting algorithms or other fields of algorithms requires building a more complex decision tree that is trained to classify those algorithms. This is left for future work. An algorithm that does not belong to the set of the five sorting algorithms will be falsely classified as one of these algorithms. We will discuss this in detail when presenting the results in Section 6.

4. EXPERIMENT

Course description

Aari was tested in a first year Data Structures and Algorithms course. The course is a theoretically-oriented, programming language independent follow-up course for CS1, lectured concurrently with our CS2 course, and directed for CS majors. However, at the beginning of this course, some programming assignments were given to the students to make sure they achieve reasonable practical way of implementing algorithms. CS1 was taught in Java, thus the programming exercises were submitted in Java as well.

125 students enrolled in this course. Most of them participated this experiment. We will come back to these figures in Section 6. We collected student submission for a sorting algorithm assignment in two phases.

Data collection

In the very first week (among other assignments), the students were given an assignment to design and implement an algorithm that sorts an array of integers into ascending order. They were asked to describe the algorithm verbally, and implement a method that would conform to the following Signature 1. The implementation was submitted to our automatic assessment system.

```
public static int[] mySort(int[] data)                (1)
```

Sorting algorithms were lectured in week 3. In the consequent weekly exercises, the students were reminded about their previous sorting algorithm design and implementation in week one. We asked them to analyze which of the lectured sorting algorithm their design resembled the most, and describe how the implementation was different from the course material. In addition, they were asked to implement and submit one of the algorithms described in the course material. Presumably one that is more efficient than the previous submission. Finally, they were asked to analyze and compare the two implementations with each other.

Each submission was tested automatically in an automatic assessment system that gave feedback about the correctness of the solution (black-box testing only). There was 1 submission in the first phase and 1 submission in the second phase that did not pass the tests. We analyzed only those implementations that passed the tests (112 implementations in the first phase and 80 implementations in the second phase), thus we were reasonably sure that the implementations were error-free.

Data preparation

Aari was used to classify the students' solutions later. Thus, this time the students did not receive any feedback about the white-box testing phase.

For the analysis purposes, we added each student's user name as a prefix to the class name and the aforementioned method name in his/her submitted programs. This helped us find the students implementations in the first and second round from the database. A few implementations used *enhanced for* statement introduced in Java 5.0 for iteration through collections and arrays. Aari does not support this form of for statement and thus these for statements were replaced with the ordinary ones.

5. CATEGORIZING ALGORITHMS

The sorting assignments did not ask student to implement any specific algorithm, thus we got a wide variety of submissions. We analyzed the submitted algorithms and categorized them manually as well as automatically. We needed the results of the manual analysis in order to assess the accuracy of Aari. Furthermore, we need the manual analysis to see what kind of variations of sorting algorithms students implement. This analysis could also reveal what kinds of misconception they have related to sorting algorithms. We need these observations in order to further develop Aari to give appropriate feedback to students about their implementations.

It was straightforward to categorize most of the submitted implementations. However, it was difficult to fit some of the implementations into any type of standard algorithms. In the following, we briefly discuss what types of student submissions we considered as standard algorithms in our analysis. Based on this discussion, we will introduce two categories that we call "Inefficient variations" and "Others", and explain how the implementations of these categories differ from standard algorithms. To keep the number of the main categories small, we included the few implementations of Shellsort and Heapsort as a subcategory of "Others". The aim of the following discussion is not to give an exact definition to the algorithms, but rather outline and emphasize the essential features of the algorithms that were used as a guideline in the categorization of the student implementations. Although some of the remarks may seem obvious, expressing them explicitly helps to understand the reasoning behind the categorization.

5.1. Bubble sort

Implementation of an in-place Bubble sort includes two nested loops and a swap operation performed in the inner loop. An essential feature of Bubble sort is that the two items compared in each pass are adjacent. This feature is emphasized by all the descriptions of Bubble sort we studied, including textbooks and articles (see, e.g., one of the earliest description in [Friend 1956], where Bubble sort is referred to as "Exchanging Sort", as also explained in [Astrachan 2003]). Some sources introduce an optimized version of Bubble sort where a boolean value is used as a one-way flag to indicate whether a swap is performed in the inner loop. If no swaps are performed, the list is sorted, and the algorithm can terminate. We consider both implementations, with or without a one-way flag, to be Bubble sorts because of the history of sorting algorithms. In his study of Bubble sort, Astrachan concludes that "Perhaps early concerns with allocating registers and using memory led to the adoption of bubble sort which in its primitive form (no early stopping) requires no storage other than the array and two index variables." In addition, there is no point to separate these two versions of Bubble sort (e.g., the non-flag version to be included in the Inefficient variation category) as neither of these are recommended in textbooks.

5.2. Insertion sort

Implementation of an in-place Insertion sort consists of two nested loops and a shift operation performed by the inner loop. Implementations that use swap operation in the inner loops instead of shift are surprisingly more common than one could expect. Even some textbooks introduce Insertion sort algorithm with swap operation; see for example [Dale et al. 2002] and [Shaffer 1998]. [Sedgewick 2002] gives an example of both versions and introduces the version with shift operation as an improved version of the one that uses swap.

However, we do not consider implementations with swap (which we call “Insertion sort with swap”) as a pure Insertion sort in our analysis, but rather as a variation of it. We discuss this variation in more detail later when presenting these “Inefficient variations”. In addition, it should be noted that in Insertion sort, the items of the given list are handled in a successive manner, that is, the second item is compared with the first item, the third item is compared with the second and the first item, and so on (see, e.g., [Friend 1956; Knuth 1998]). We will use this definition to differentiate between the implementations of Insertion sort and the other non-recursive borderline cases.

5.3. Selection sort

Implementation of an in-place Selection sort consists of two nested loops as well. In the inner loop, the position of the smallest (or largest) item is stored in each pass and in the outer loop, this element is exchanged with the item pointed by the loop counter of the outer loop. In addition to the position of the smallest (or largest) item, some students’ implementations store the value of the item as well and use it in the swap operation in the outer loop. We consider all these kinds of implementations to be Selection sorts.

On the other hand, some implementations exchange the smallest (or largest) item found so far during one single pass in the inner loop with the item pointed by the loop counter of the outer loop and thus may perform several unnecessary swap operations within the inner loop during one pass instead of storing the position of the wanted item and exchange it only once in the outer loop (see Figure 2). We call these implementations “Selection sort with inner loop swap” and consider them as a variation of Selection sort, and similar to the Insertion sort variations, these are considered to be a subtypes of the category “Inefficient variations”.

5.4. Mergesort

Implementations of the submitted Mergesorts followed the typical algorithm described in textbooks: dividing the given array of items into two halves, sorting each half using recursion and merging the sorted halves into one sorted array. Although the submitted implementation codes varied greatly, for example, in terms of the number of the used auxiliary arrays, we did not discern clear variations within the implementations.

5.5. Quicksort

The submitted implementations of in-place Quicksort followed the same basic idea: partitioning the given array and calling the method recursively for both partitions. Some implementations selected the pivot item from the left or right end and some from the middle of the given array. Moreover, some implementations used extra array(s). However, we did not divide the implementations to different subcategories based on the pivot selection strategies or using extra arrays.

5.6. Inefficient variations

A common feature in some of the Insertion and Selection sort variations was that they did unnecessary swaps in the inner loop, and thus are somewhat inefficient. There is always room for (quite an obvious) optimization. We selected the name “Inefficient variations” to reflect this fact. Although Bubble sort and its variations can also be included in this category, we classified it as its own category as it is a standard algorithm that is widely covered in textbooks. In addition, in case of Insertion and Selection sort, at least some of the implementations are a consequence of misconception related to the original algorithm. Thus, in order to give automatic feedback of such misconception in the future, we need to develop an instrument for recognizing these variations.

```
int i, j, temp;
for (i = 0; i < table.length-1; i++) {
    for (j = i+1; j < table.length; j++) {
        if (table[i] > table[j]) {
            temp = table[i];
            table[i] = table[j];
            table[j] = temp;
        }
    }
}
```

Fig. 2. An example of a variation of a Selection sort in the students’ implementations that uses swap in the inner loop

Figure 2 shows the most common variation within the students’ implementations. As can be seen from the figure, the iterations of the outer and inner loops, the comparison of the two items and exchanging the items are different from those discussed above for Selection and Insertion sorts. Some of these variations may store the index of the smallest item, but they leave it unused and do the swap in the inner loop. While classifying algorithms, we have ignored all unused variables, and consider the algorithm to be the one without those code lines including redundant code.

The second common variation in student implementations are variations of Insertion sort that use swap instead of shift.

5.7. Others

This category consists of the implementations that cannot be included in the algorithms discussed above. The following subcategories can be discerned.

The first subcategory is called “Other standard algorithms” and includes the implementations of Shellsort, Heapsort and hybrid Quicksort-Selection sort algorithm. These implementations were quite similar to those discussed in textbooks. and we will not discuss them further here.

The second subcategory, which we have called “Other variations”, consists of those algorithms that have an established name, but are not usually covered in textbooks nor taught in a CS class because of their pathological or poor worst and average case performance. These include Bogosort (a sorting algorithm that randomly shuffles the items of a list until they are in order), Bozo sort (a random sort like Bogosort, but picks two items at random and exchanges them until the list is sorted) and Gnome sort (a single loop algorithm that exchanges the two out-of-order adjacent items and checks the items which are positioned before the exchanged items and continues this until the list is sorted). The variations of Gnome sort are the most common of these types of variations in the students’ implementations. Figure 3 shows an example implementation of a Gnome sort.

The third subcategory includes “Own method” implementations. This category consists of the solutions that we found difficult to include in any category or subcategory

```
for(int i=1;i<table.length;i++){
    if(table[i]<table[i-1]){
        int tmp = table[i];
        table[i] = table[i-1];
        table[i-1]= tmp;
        if(i>1)
            i = i-2;
    }
}
```

Fig. 3. An example of the variations of a Gnome sort in the students' implementations

discussed above. Typical solutions try to follow an idea from some of the in-place algorithms, but use an unnecessary auxiliary array. By this we mean an array that is used tightly coupled with the input array in order to do the sorting. However, if the items are just copied into an auxiliary array, and the sorting is done in this new array without the need of the original one, we did not consider this alone to constitute an Own method. In addition, other common characteristics in Own methods are the excessive use of loops and variables. For example, the algorithm in Figure 4 inserts each item from the original array `data` into the appropriate position in an auxiliary array `numbers`, and afterwards moves the items toward the end of the auxiliary array one by one in the third loop in line 11. Although it works similar to the idea of insertion sort, it is difficult to consider it to be a variation of insertion sort. Not because of the use of the auxiliary array, but because of three loops used in the implementation.

Finally, the fourth subcategory contains solutions in which no algorithm was designed nor implemented by the student. That is, the solutions used the implementation provided by the Java standard library.

```
01 int[] numbers = new int[data.length];
02 int i, j, lastRead, tmp, tmp2;
03 for(i = 0; i < data.length; i++){
04     lastRead = data[i];
05     for(j = 0; j <= i; j++){
06         if(j == i)
07             numbers[j] = lastRead;
08         if(numbers[j] >= lastRead){
09             tmp = numbers[j];
10             numbers[j] = lastRead;
11             for(int m = j+1; m <= i; m++){
12                 tmp2 = numbers[m];
13                 numbers[m] = tmp;
14                 tmp = tmp2;
15             }
16             break;
17         }
18     }
19 }
```

Fig. 4. An example of an "Own Method" in the students' implementations

6. RESULTS

We analyzed the implementations manually and automatically. In the following, we first present the results of the manual analysis and then the results of the automatic recognition of the implementations by Aari.

Results of manual analysis

112 students submitted their solutions to the sorting problem in the first round and 80 students in the second round (all the students who submitted a solution in the second

round also did so in the first round). Figure 5 shows the distribution of the algorithms submitted by the students in each round and compares the number of each algorithm in the first and second round. In the first round, most of the students implemented Selection sort (30 students, 27 percent of all the submissions of the first round). The second most common implemented algorithm is Bubble sort (25 students, 22 percent). Not surprisingly, the implementations of the more efficient sorting algorithms are not common in the first round, since the deadline for this round was before the lectures on sorting algorithm started. For example, there were only 4 Mergesort (4 percent) and 2 Quicksort (2 percent) implementations among the submissions. On the other hand, Quicksort and Mergesort algorithms are the most common implementations in the second round (32 and 16 implementations, i.e., 40 and 20 percent, respectively). Again, as one could expect, “Inefficient variations” and “Others” implementations are much more common in the first round (20 students, 18 percent and 23 students, 21 percent, respectively) than in the second round (3 students, 4 percent and 10 students, 13 percent, respectively).

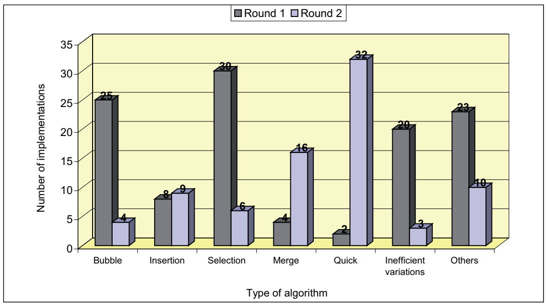

Fig. 5. All the students’ implementations of sorting algorithms in the first and second round

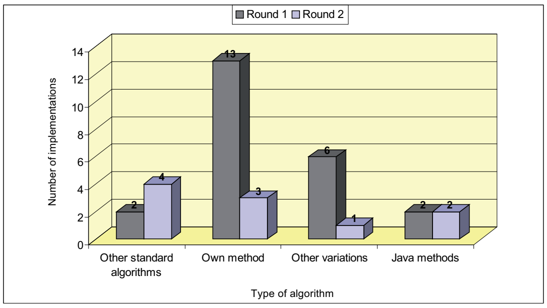

Fig. 6. Students’ implementations of sorting algorithms classified as “Others” in Figure 5.

Table II summarizes these results and shows the number (the first number in the columns) and percentage (the second number in parentheses) of the implementations of each submitted algorithm. The results are presented separately for the first round, second round and in total. Note that the percentage show the results with respect to the number of algorithm in the corresponding round. For example, the number of the implementations of Bubble sort algorithm in the first round is 25 (i.e. 22 percent of

Table II. The number and percentage of each type of sorting algorithm submitted by the students in the first and second round

Algorithm	Round 1 (%)	Round 2 (%)	Total
Bubble sort	25 (22)	4 (5)	29 (15)
Insertion sort	8 (7)	9 (11)	17 (9)
Selection sort	30 (27)	6 (8)	36 (19)
Mergesort	4 (4)	16 (20)	20 (10)
Quicksort	2 (2)	32 (40)	34 (18)
Inefficient variations	20 (18)	3 (4)	23 (12)
Others	23 (21)	10 (13)	33 (17)
Total	112	80	192

Table III. The number and percentage of sorting algorithms categorized as "Others" in the first and second round

Algorithm	Round 1 (%)	Round 2 (%)	Total
Other standard algorithms	2 (9)	4 (40)	6 (18)
Own method	13 (56)	3 (30)	16 (49)
Other variations	6 (26)	1 (10)	7 (21)
Java methods	2 (9)	2 (20)	4 (12)
Total	23	10	33

the overall data in the first round which is 112), and in the second round 4 (which is 5 percent of the overall implementations submitted in the second round, i.e., 80).

Figure 6 illustrates the number of different subcategories in "Others". The largest subcategory in the first round is "Own method" (13 implementations), whereas the largest subcategory in the second round is "Other standard algorithms" (4 implementations).

Distribution of different subcategories of "Others" (in number and percentage) is shown in Table III. As in Table II, the numbers in parentheses show the percentage of the algorithms in the corresponding round. For the column Total, the percentage show how frequent each subcategory category is with respect to the all implementations of "Others" in the first and second round.

We also investigated the performance of each student in the two rounds to see how many of them improved their solutions. From the 80 students who submitted a solution in the second round, two students implemented the same algorithm as in the first round (different implementations of Selection sort). 19 students implemented a Bubble, Selection or Insertion sort in the first round and changed to another algorithm from this group on the second round (includes also the variations of these algorithms). Three students who had implemented an efficient algorithm in the first round did so also in the second round. Two Mergesort implementations in the first round were changed to the implementation provided by the Java standard library and an implementation of an Insertion sort with swap in the second round. One student changed from "Own method" to Java standard library and one from Bubble sort to Bozo sort. The rest of the students (i.e., 52 students) implemented a more efficient algorithm in the second round than in the first round.

Results of automatic recognition

The results of evaluation of the accuracy of Aari are depicted in Figures 7 (for the first round) and 8 (for the second round). Aari has a mechanism to recognize five sorting algorithms: Bubble sort, Insertion sort, Selection sort, Mergesort and Quicksort. As can be seen from the figures, these algorithms are generally recognized very accurately in both rounds. For example, all the implementations of Selection sort and Quicksort algorithm are correctly recognized. However, not surprisingly, Aari does not perform that well with the algorithms that it has not been trained to recognize. All the implementations of “Other standard algorithm”, that is, 1 Shellsort, 4 Heapsorts and 1 hybrid algorithm of Quicksort and Selection sort, as well as the implementations of “Other variations” and “Java methods” subcategories are recognized falsely. From the 16 implementations of “Own method” category, 4 implementations of Selection sort and 1 implementation of Quicksort (which were categorized as Own method because they do not sort in place) were recognized as Selection sort and Quicksort.

The number and percentage of correctly and falsely recognized implementations for each type of submitted algorithm are shown in Table IV, for each round separately, as well as for all the implementations in total. Again, it should be noted that the percentage show the results with respect to the number of specific algorithms in the corresponding round. For example, the number of the implementations of Bubble sort algorithm in the first round is 25, from which 24 algorithms are correctly recognized, that is, 96 percent of the 25 algorithms.

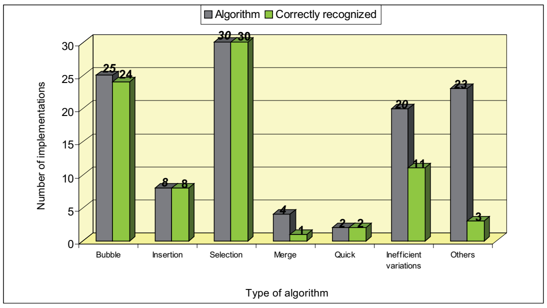

Fig. 7. The number of different sorting algorithms implemented by the students in the first round (gray column) and recognized by Aari (green column)

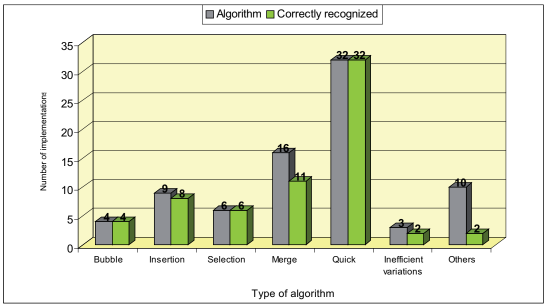

Fig. 8. The number of different sorting algorithms implemented by the students in the second round (gray column) and recognized by Aari (green column)

In the case of category “Inefficient variations”, 13 out of 23 are recognized correctly (i.e., as Selection and Insertion sorts) in total, that is, 57 percent of all the “Inefficient variations” in both rounds. From the 20 “Inefficient variations” in the implementations of the first round, there were 12 variations of Selection sort and 8 variations of Insertion sort. In the second round the numbers were 1 and 2, respectively. Furthermore, the subcategory “Other variations” in “Others” included 1 variation of Bogosort and 5

Table IV. The number and percentage of each type of sorting algorithm correctly recognized by Aari. Columns Round 1 and Round 2 show the number of submitted algorithms

Algorithm	Round 1	Correct (%)	Round 2	Correct (%)	Total	Correct (%)	False (%)
Bubble sort	25	24 (96)	4	4 (100)	29	28 (97)	1 (3)
Insertion sort	8	8 (100)	9	8 (89)	17	16 (94)	1 (6)
Selection sort	30	30 (100)	6	6 (100)	36	36 (100)	0 (0)
Mergesort	4	1 (25)	16	11 (69)	20	12 (60)	8 (40)
Quicksort	2	2 (100)	32	32 (100)	34	34 (100)	0(0)
Inefficient variations	20	11 (55)	3	2 (67)	23	13 (57)	10 (43)
Others	23	3 (13)	10	2 (20)	33	5 (15)	28 (85)
Total	112	79 (71)	80	65 (81)	192	144 (75)	48 (25)

Table V. The confusion matrix showing the classification accuracy of Bubble sort, Insertion sort, Selection sort, Mergesort and Quicksort implementations

Class	**Bubble**	**Insertion**	**Selection**	**Merge**	**Quick**
Bubble sort	28	1	0	0	0
Insertion sort	1	16	0	0	0
Selection sort	0	0	36	0	0
Mergesort	4	0	0	12	4
Quicksort	0	0	0	0	34

variations of Gnome sort in the first round, and 1 variation of Bozo sort in the second round.

We use a *confusion matrix* to look at the accuracy of the classification on the overall data. A confusion matrix is an $N \times N$ matrix, where each instance I_{ij} indicates the instance that belongs to class I_i, but is classified as class I_j [Tan et al. 2006]. As the decision tree used for classifying algorithms has mechanism to recognize Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort algorithms (see Figure 1), we use a 5×5 confusion matrix to see how accurately the implementations of these algorithms were recognized. The resulted confusion matrix is shown in Table V. Note that the variations of Selection and Insertion sort are not included in the confusion matrix. As the confusion matrix shows, from the 20 instances of Mergesorts in the first and second round, 4 are recognized as Bubble sorts and 4 as Quicksorts. This is because either the falsely classified implementations of the Mergesort are falsely detected as non-recursive or as tail recursive algorithms, respectively. Other four types sorting algorithms are recognized with high accuracy.

Finally, we can present the accuracy of the decision tree of Figure 1 for the five types of sorting algorithms in terms of true positive (TP, correctly recognizing an implementation that belongs to the target set), false negative (FN, falsely recognizing an implementation of the target set as another one) and false positive (FP, falsely recognizing an implementation which does not belong to the target set, as a member of the target set) metrics. Moreover, based on these metrics, we can also express the following metrics for the five sorting algorithms: true positive rate (TPR, the proportion of the positive case implementations that are correctly recognized, i.e., *TPR = TP/(TP + FN)*), false negative rate (FNR, the proportion of the positive case implementations that are not recognized correctly, i.e., *FNR = FN/(TP + FN)*) and precision (the proportion of the actual positive case implementations to all the implementations recognized as positive, i.e., *precision = TP/(TP + FP)*). Table VI summarizes these results. The implementations of Bubble sort algorithm have the largest number of FP cases (19 samples), and as a result, they have the poorest result of precision. This is because the implementations of Bogosort, Bozo sort and Gnome sort as well as the implementations that use standard Java library methods are recognized as Bubble sort.

Table VI. The value of true positive (TP), false negative (FN), false positive (FP), true positive rate (TPR), false negative rate (FNR) and precision for the implementations of the five types of sorting algorithms

Class	Total	TP	FN	FP	TPR	FNR	Precision
Bubble sort	29	28	1	19	96.6%	3.4%	59.6%
Insertion sort	17	16	1	4	94.1%	5.9%	80%
Selection sort	36	36	0	6	100%	0%	85.7%
Mergesort	20	12	8	0	60%	40%	100%
Quicksort	34	34	0	6	100%	0%	85%

7. DISCUSSION

We first discuss our observations on manual analysis of the students' implementations and see what they suggest in terms of further developing Aari as a white-box testing instrument. This is followed by a discussion on the results of evaluating Aari in its current state as an automatic assessment tool to check student submissions.

Analyzing student implementations

As Knuth puts it, "the classification of sorting methods into various families such as "insertion," "exchange," "selection," etc., is not always clear-cut." [Knuth 1998]. This is especially true when classifying novices' implementations. For example, we have categorized the algorithm depicted in Figure 2 as a variation of a Selection sort because of the steps taken in the two nested loop. In this type of variation, the notion of selection is embedded in the swaps (i.e., selecting the value lesser than that shown by the loop counter of the outer loop and swapping them). However, one can argue that since the position of the min/max element is not explicitly stored or used, it cannot be considered as a Selection sort. This is a perfectly legitimate argument. The question remains, however, what algorithms does it represent? For example, it cannot be regarded as a Bubble sort neither, because unlike in Bubble sort, the two compared items are not adjacent (see the features of a Bubble sort outlined in Section 5. In fact some websites name this algorithm as an Exchange sort, while others use this name interchangeably with Bubble sort). We have introduced the category "Inefficient variations" for these kind of variations that do unnecessary swaps in the inner loop and should not be used. It clarifies the categorization.

One objective of this study is to investigate what types of misconceptions students have about sorting algorithms (note that since the students had no formal instructions in the field of sorting algorithms when doing the assignment of the first round, misconception in this round refers to what they have not fully understood when reading/thinking about sorting problem, rather than misunderstanding what they have been taught by the teacher). We therefore did not ask the students to implement a particular sorting algorithm, because this could have resulted in submitting "perfect implementations", presumably by getting help from some sources, and thus would not have served our purposes. Our goal is to apply the gathered information to further develop Aari to give feedback to students about these misconceptions and to suggest how they can improve their implementations.

In the analysis, we found many misconceptions/shortcomings in student work. Two of these misconceptions relate to the implementations of Insertion sort and Selection sort algorithm: these implementations use unnecessary swaps that make them inefficient. We have classified these implementations as "Inefficient variations" and as Table II shows, these implementations constitute 12 percent of the total submissions in the first and second round. We can give feedback to the students to avoid these unnecessary swaps. Other common shortcomings include using unnecessary variables, assignment statements and auxiliary arrays. Although we did not consider the issues like using auxiliary arrays (see Section 4), etc., as a basis to classify the algorithm in question as a variation, it is clearly beneficial to give feedback to the students on these shortcomings, as well.

In case of Bubble sort implementations, in addition to giving feedback on the inefficiency of the algorithm in general, we can provide feedback for optimization in the case of the implementations that do not use a boolean variable as a one-way flag to check whether to continue iterations (surprisingly, more than half of the all submitted Bubble sort implementations [i.e., 55%] use this optimization, although most textbooks and other sources do not discuss it widely). For the more efficient sorting algorithms such as Quicksort, an improvement suggestion can include using more efficient pivot selection strategy as well as not using unnecessary auxiliary array. From the 32 implementations of Quicksort in the second round, 12 used the leftmost or rightmost element as the pivot, while 22 implementations used the middle element. In the example of the handout of the course, the rightmost element of the given array is selected as the pivot. However, more efficient strategies for selecting the pivot are discussed.

Implementations of efficient algorithms were much more common in the second round than in the first round (see Figure 5 and Table II). As an example, the number of Bubble sort implementations was much higher in the first round than in the second round (25 and 4, respectively), while the number of Quicksort implementations increased significantly in the second round (2 in the first round and 32 in the second). Similarly, the students submitted much more implementations of “Own method” and “Other variations” of category “Others” in the first round than in the second round. (See Figure 6 and Table III). A straightforward conclusion would be the impact of the instruction, but drawing any conclusion requires further investigation of the material (students’ descriptions, analyses, etc.).

In Section 2, we discussed the study of Simon et al. [Simon et al. 2006] on students’ knowledge of sorting algorithms before they start a CS1 course. In that study the students were asked to describe a sorting algorithm in English, whereas the assignment in our study was an implementation assignment. Furthermore, our subjects were students starting a course on data structures and algorithms and had completed a CS1 course (although sorting algorithms are not covered in the CS1 course). However, we can find the following differences and similarities in the observations of our study in the first round compared to the Simon et al. findings. Unlike the observation of Simon et al., all of the implementations in our study treated numbers as primitive types rather than strings of digits. Selection sort and Bubble sort were the most popular algorithm in our data, while in the Simon et al. study, the most popular algorithms were Selection sort and Insertion sort. Furthermore, we had a few submissions that implemented more efficient sorting algorithms (such as Mergesort and Quicksort), while there were no descriptions of the more efficient algorithms in the Simon et al. study. With regard to the similarities, like in the Simon et al. study, implementation of Selection sort algorithm was the most popular solution in our study as well. Finally, Simon et al. conclude that although most of the beginners treated numbers as strings of digits, they were “very likely to provide a correct solution to the problem” [Simon et al. 2006]. Based on our observations, we can also conclude that the students were able to submit working solutions to the sorting problem before being formally taught by an instructor.

Validating Aari

Another reason for giving a general implementation task to the students rather than asking them to implement a particular sorting algorithm was to test the performance of Aari more comprehensively. The five algorithms that Aari is trained to recognize are recognized with high accuracy (see Table VI). Except the implementations of Mergesort, the implementations of other four algorithms are recognized with more than 94% true positive rate (the average accuracy for all the five algorithms is about 90%). Aari is not trained to recognize the other algorithms, hence the results of recognizing the algorithms other than these five types of sorting algorithms are poor. However, a mechanism for recognizing other types of sorting algorithms as well as other fields of algorithm can be added to Aari to address this limitation.

Category "Inefficient variations" includes the implementations of Insertion sort with swap and Selection sort with inner loop swap. The decision tree of Figure 1 does not recognize these variations as their own class, but rather as Insertion and Selection sorts. Therefore, we have considered the implementations of this category as recognized correctly, if they are recognized as Insertion and Selection sorts, respectively. Based on the category emerged from the data, we will further develop Aari to recognized these variations as their own types. This will make it possible to give more accurate feedback on students' implementations.

From the implementations of category "Inefficient variations", all the variations of Insertion sort (10 implementations) were recognized correctly, while most of the variations of Selection sort (10 out of 13 implementations) were recognized falsely. The reason is that as the decision tree of Figure 1 illustrates, in order to recognize an implementation as an Insertion sort, the implementation must have the characteristic IITO (Inner loop counter Initialized To Outer loop counter). Like the implementations of a "pure" Insertion sort, all of the variations of Insertions sort had this characteristic and thus are recognized as Insertion sort. However, an implementation of a Selection sort is recognized by its MWH (most-wanted holder) role. While some of the implementations of the variations of Selection sort do store the position (and some the value as well) of the so far found smallest/largest item (although leave it unused and do unnecessary swaps in the inner loop), most of them do not store it. For those implementations that store the position, the role detector detects a MWH role and thus they are recognized as a Selection sort. For the rest, no MWH role is detected and as the result, they are not recognized as a Selection sort. All of these latter implementations are recognized as an Insertion sort, since they have the characteristic IITO.

These variations can be recognized more accurately and these kinds of errors can be eliminated by further developing Aari and integrating schema recognition mechanism into it. Schema recognition will, as an example, detect a swap operation within the inner loop and result in a correct recognition of the implementations of those variations of Selection sort that do unnecessary swaps in the inner loop. To summarize, Aari performs much better with the implementations of the second round (accuracy of 81%) than with the implementations of the first round (accuracy of 71%). This is because the first round includes more implementations of "Own method" and "Inefficient variations", in other words, the implementations of the second round are closer to standard algorithms.

Implications for education

Sorting algorithms is one core topic in a data structures and algorithms course. Even though in real-life programming projects sorting can often be implemented using library functions, all computer science students should learn at least some of key simple algorithms, such as Insertion or Selection sort, as well as more efficient algorithms like Quicksort and Mergesort. Equally important is understanding their run time performance analysis, and which aspects of the algorithms are the key factors for performance.

We do not claim that covering such a wide set of sorting methods and their variations, in such details as we have discussed in this paper, is worthwhile education considering the totality of topics that students need to learn on their programming courses. We, however, want to emphasize other aspects that our analysis revealed, and which should be addressed in general programming education. Inefficient versions of simple algorithms can be used as cases which demonstrate several aspects how the code could and should be made more clear and efficient. We can discuss topics such as: Why we should aim at keeping code simple? Why should we avoid using extra variables and operations (like swaps in this context) that could be avoided by small refinements of the algorithm? Is there any sense of tuning our algorithms manually, when current compilers often do similar optimizations? Or, what is the role code readability vs. efficiency? With such discussion we could urge our students to take a critical view of their code.

Automating the feedback on algorithms in the context of automatic assessment with the Aari system has its pros and cons. For a teacher, such information would be a welcome assistance when checking manually students' code. This would allow giving better personal comments for the students and pick up interesting examples to discuss with students. In such cases, where students have been requested to implement a specific algorithm, the automatic analysis/feedback would allow the teacher to concentrate only on such cases which did not fit the specification, instead of checking all codes. From students' perspective, getting some feedback of algorithm implementation would provide new information and point of view to look when compared with what existing automatic assessment tools can provide. As the analysis is, however, not foolproof, the feedback must be phrased so that it suggests something the student should look at, instead of making a summative evaluation that something is right or wrong. The current evaluation results, however, indicate that the accuracy of the analysis is rather high, which is promising.

Points of Validity

The results of the analyzed data set are from a specific course, and may not generalize to other settings. In our case, sorting algorithms are not covered in our CS1 course, while in many other institutes at least some basic methods are covered. Available course resources and textbooks might have a major effect on the results, if a similar open task of implementing any sorting algorithm would be set up. What is interesting in our case is that so many students did not solve the problem by Googling and copying available code from the net but only implemented some of their own solutions.

8. CONCLUSION AND FUTURE WORK

In this paper, we have validated Aari, the previously introduced instrument for algorithm recognition [Taherkhani 2011] with authentic students' implementations. For the five types of sorting algorithm implementations that Aari has been trained to recognize (see Figure 1), the results are very promising (see Table II and Figures 7 and 8). Furthermore, more than fifty percent of the variations of these algorithms are also recognized correctly. However, as expected, all of the other types of algorithms are recognized falsely.

Checking students' work in large courses is a difficult and time-consuming task. Instructors use automatic tools to assess the correctness of these works. Aari can provide help in checking that the works implement the required algorithm. As the recognition method described in section 3 is statistical by nature, it may not recognize algorithms with 100% of accuracy. However, human inspectors make errors as well and thus reasonable accuracy could be sufficient. Furthermore, instructors can use Aari as a semi-automatic tool as well: it is of great help if the major part of the assignments are correctly recognized by Aari and an instructor has to check only the rest of them manually.

Our next plan is to further develop Aari as a white-box instrument. Based on the data and analyses presented in this study, we have observed a set of variations of standard algorithms that can be considered in the future development of Aari. This mainly includes the category “Inefficient variations” introduced in Section 5. The variations of standard algorithms, especially Insertion and Selection sort, were very common within the students’ implementations and this suggests that giving feedback on these kind of implementations can be beneficial and help students understand the target algorithm better. In addition, feedback on avoiding unnecessary variables, assignments statements, auxiliary arrays, etc., that also improve students’ general programming skills can be given to them. These features make Aari a helpful white-box instrument that along with checking functionality (black-box testing), not only assist instructors in their assessment task, but also help students in learning task by providing appropriate feedback on their misconceptions and bad solutions.

To implement the above plan, our next step of future work is to integrate schema recognition mechanism into Aari. This will help us not only identify the algorithm-related code from non-relevant application data and select it for further analysis, but also detect those features of the code that in its current state, Aari is not capable of detecting. As an example, in order to differentiate between an implementation of a standard Selection sort and an implementation of a Selection sort with inner loop swap, it is essential to detect whether a swap operation is performed in the inner loop or in the outer loop. Schema recognition mechanism will make such detections possible.

We have demonstrated the performance of Aari in the case of sorting algorithms. As another direction of future work, we need to cover other fields of algorithms and show the performance of Aari with a more comprehensive set of different algorithms using the appropriate empirical experiments. In addition, we believe that students have similar misconceptions on other fields of algorithms as well (see, e.g., [Seppälä et al. 2006]). We should conduct similar studies as presented in this paper to find out these misconceptions and further develop Aari to give feedback also on them. Finally, we need to evaluate students’ responses to the added feedback on their algorithms.

9. ACKNOWLEDGMENT

We would like to thank Lasse Hakulinen for helping us with collecting the student submissions from the automatic assessment system.

REFERENCES

ALA-MUTKA, K. 2005. A survey of automated assessment approaches for programming assignments. *Computer Science Education 15,* 2, 83–102.

ALA-MUTKA, K. AND JÄRVINEN, H.-M. 2004. Assessment process for programming assignments. In *Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies (ICALT’04), Joensuu, Finland, August 30–September 01*. 181–185.

ASTRACHAN, O. 2003. Bubble sort: An archaeological algorithmic analysis. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education(SIGCSE ’03), Reno, Nevada, USA, February 19–23*. ACM New York, NY, USA, 1–5.

BISHOP, C. AND JOHNSON, C. G. 2005. Assessing roles of variables by program analysis. In *Proceedings of the 5th Baltic Sea Conference on Computing Education Research, Koli, Finland, 17–20 November*. University of Joensuu, Finland, 131–136.

DALE, N., JOYCE, D., AND WEEMS, C. 2002. *Object-Oriented Data Structures Using Java* Second Ed. Jones and Bartlett Publishers.

EDWARDS, S. H. 2003. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, California, USA, 26–30 October*. ACM, New York, NY, USA, 148–155.

ELENBOGEN, B. S. AND SELIYA, N. 2007. Detecting outsourced student programming assignments. *Computing Sciences in Colleges 23,* 3, 50–57.

FRIEND, E. H. 1956. Sorting on electronic computer systems. *Journal of the ACM 3,* 3, 134–168.

GERDT, P. AND SAJANIEMI, J. 2006. A web-based service for the automatic detection of roles of variables. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, Bologna, Italy, 26–28 June*. ACM, New York, NY, USA, 178–182.

HIGGINS, C., SYMEONIDIS, P., AND TSINTSIFAS, A. 2002. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, 24–26 June*. ACM, New York, NY, USA, 46–50.

JOY, M., GRIFFITHS, N., AND BOYATT, R. 2005. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing 5,* 3, 1–28.

KNUTH, D. E. 1998. *The Art of Computer Programming: Sorting and Searching* Second Ed. Vol. 3. Addison-Wesley, New Jersey, USA.

MCCABE, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 308–320.

SAIKKONEN, R., MALMI, L., AND KORHONEN, A. 2001. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*. ACM Press, New York, Canterbury, UK, 133–136.

SAJANIEMI, J. 2002. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments, Arlington, Virginia, USA, 3–6 September*. IEEE Computer Society Washington, DC, USA, 37–39.

SAJANIEMI, J., BEN-ARI, M., BYCKLING, P., GERDT, P., AND KULIKOVA, Y. 2006. Roles of variables in three programming paradigms. *Computer Science Education 16,* 4, 261–279.

SAJANIEMI, J. AND PRIETO, R. N. 2005. Roles of variables in experts programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*. University of Sussex, U.K, 145–159.

SEDGEWICK, R. 2002. *Algorithms in Java, Parts 1-4* Third Ed. Addison-Wesley.

SEPPÄLÄ, O., MALMI, L., AND KORHONEN, A. 2006. Observations on student misconceptions – a case study of the build-heap algorithm. *Computer Science Education 16,* 3, 241–255.

SHAFFER, C. A. 1998. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall Inc, New Jersey, USA.

SIMON, B., CHEN, T.-Y., LEWANDOWSKI, G., MCCARTNEY, R., AND SANDERS, K. 2006. Commonsense computing: what students know before we teach (episode 1: sorting). In *Proceedings of the 2006 International Workshop on Computing Education Research (ICER 06)*. 29–40.

TAHERKHANI, A. 2011. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *The Computer Journal 54,* 11, 1845–1860.

TAHERKHANI, A., KORHONEN, A., AND MALMI, L. 2011. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *The Computer Journal 54,* 7, 1049–1066.

TAN, P.-N., STEINBACH, M., AND KUMAR, V. 2006. *Introduction to Data Mining*. Addison-Wesley, USA.

Aalto-ST 4/2012

ISBN 978-952-60-4526-9 (pdf)
ISSN-L 1799-4896
ISSN 1799-490X (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi