Removed Stuff

Material not included in Master's Thesis

9.1.2020

Artturi Tilanterä

Abstract

This document contains writings which originally were part of Artturi's Master's Thesis, but were then removed. Here are additional notes, ideas and calculations. Some of them might be useful for future research.

1 Introduction

1.1 Learning analytics

Learning analytics has background in more general term **analytics**, which is the use of machine learning and statistics to generate information that enhance decision-making [5]. An early academic paper from year 2004 discusses learning analytics in relation to analytics in a business intelligence context [2]: organisations measured their training events using end-of-class questionnaires, skills-needs assessments, and instructor questionnaires. The typical reasons for learning analytics were to present the value of the training to the organisation, or to indicate the quality of the training.

1.2 Learning management system

A **learning management system** (LMS) is web software which provides online courses to students and tools for teachers to create and publish these courses and monitor students' performance. It handles learning content and student registration, and may also have assignments and communication possibility between students and teachers. [3][p. 28], [11, p. 55]. A LMS is the modern tool for learning analytics.

The exercises studied in this thesis are part of an online course on data structures and algorithms, which is provided on the *A+ Learning Management System* [19]. A+ is a web service which provides the user interface for the student to view the instructor-written electronic textbooks of courses, to work with exercises and to give feedback. A+ stores information on students' enrolment and exercise submissions on courses. The course teacher can monitor the learning data and add new material and exercises [9]. From software architecture perspective, A+ is a front-end which connects to other web services. One of the is *mooc-grader*, which actually provides the HTML-based learning material, questionnaire exercises and launches *grader programs* for automatic assessment of programming exercises [20].

Algorithm simulation exercises have been used at Helsinki University of Technology, the predecessor of Aalto University, since the year 1991. Initially the exercises were not visual and interactive, but still automatically graded by a software system called TRAKLA. [14, p. 111–112]

1.3 Pedagogics of VAS and misconceptions

Examples of beginning programmers' misconceptions already from year 1983, using the imperative BASIC programming language, include understanding the assignment statement "`LET A = B + 1`" as storing an algebraic equation into memory, or confusement between the name and the value of a variable [1]. [28, p. 358–368] has listed 162 common novice programmer misconceptions, and many of them seem still valid regardless of evolution of programming languages. The assignment statement misconception is understandable: the learner tries to apply their existing knowledge about mathematics.

My personal experience and discussion with a peer student confirm the imitative problem solving strategy. Learning an algorithm is somewhat similar to learning to play a new game: there are set of detailed rules which apply to certain situations. Still one understands the

dynamics and strategic subtleties of the game after several times of playing. Similarly, understanding a new algorithm requires multiple reading passes of the pseudocode description of the algorithm and tracing the execution with some input. If the algorithm is complex, it is tempting to try to complete a VAS exercise having both a working instance of the problem with one input and the slideshow of the model solution of another instance. This might be done in hope of accidentally completing the exercise with little effort, gaining a bit of insight of the connection between a situation in the VAS exercise and the pseudocode. After all, due to the visual and dynamic nature of the VAS exercises, one might first gain intuitive knowledge and after then connect it to the formal description of the algorithm.

2 Background

2.0.1 Test data generation by symbolic execution

An attempt in generating a misconception-aware input for Build-min-heap according to IR3 and Figure **??**.

First we must choose an execution path which reaches as many states in Figure **??** as possible.

Step	State	Choice	Reason
1	1		
2	2a		
3	2b	Yes	otherwise trivial execution
4	3		
5	6		
6	7,8		
7	9	Yes	free choice
8	10		
9	14	Yes	free choice
10	15		
11	17	Yes	choose yes to recur
12	18		
13	19		
14	6		
15	7,8		
16	9	No	"Yes" was explored eariler
17	12		
18	14	No	"Yes" was explored earlier
19	17	No	"Yes" was explored earlier
20	21		
21	21		
22	2c		
23	2b	No	"Yes" was explored earlier
24	5		

Okay, that was logical. The execution path is the sequence of the states 1, 2a, 2b, ..., 2b, 5. Next we must deduce the input based on this execution path.

Because the first call to MIN-HEAPIFY recurs once, the heap must have three levels and therefore $4 \ldots 7$ elements. Let's assume HEAP-SIZE$(A) = 4$ which leads to $i = 2$ in step 2a. Now, in step 7,8, $l \leftarrow$ LEFT-CHILD-INDEX$(i = 2) = 4$ and correspondingly $r \leftarrow$ RIGHT-CHILD-INDEX$(i = 2) = 5$, as in Figure 1.

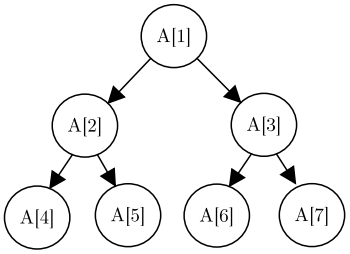


Figure 1: Array indices of a binary heap with three levels, corresponding indexing of Algorithm **??**.

2.0.2 Boundaries of input generation

My little hypothesis rant: it is impossible to generate random input which is aware of the unknown. The idea of manually preconfigured templates is like black-box testing against specification; the teacher only knows how the correct algorithm should work and what are some typical misconceptions. Input generation from symbolic execution of misconception algorithm variants is like automatic or semiautomatic white-box testing. (Or "gray-box" testing, whatever that means.) Otherwise, if we don't know what kind of logic student's mental model might have, it is just random shooting of constructed inputs and guessed corner cases in hope of actually finding something new. There is also the limitation that we don't even have an unknown computer software which could be dynamically reverse-engineered by running it again and again, but a human which runs their mental model on random input a couple of times. One VAS exercise might have 600 submissions and cannot possibly know whether there is any algorithm behind each submission. Clearly the misconception-aware input generation of VAS exercise has some analogies to software testing, but it also has many differences and reasons why some software testing practises cannot be applied.

The input requirements IR1–IR6 suggest a problem which does not have a perfect solution: how could be produce a random input which is very limited in length and still discriminates both all known misconceptions and reveal something new?

More formally: if using symbolic execution, one could derive rules for input generation for a VAS exercise, thus guaranteeing IR4 and IR5, how much is the input constrained? This is, in a way, an automatic method for producing preconfigured templates. Is it computationally cheaper than generating random inputs and validating them against IR4 and IR5? How complex constrains would the method result? Could answer set programming (software: https://potassco.org/) be used for it? How much fulfilling IR4 and IR5 does hinder the detection of new misconceptions?

2.1 Ideas for VAS input generation by symbolic execution

Here is a *draft* on how a symbolically executing input generator for a VAS exercise might work[1]. It assumes that the control flow graphs of the correct algorithm and known misconception algorithms are already generated (this can be done automatically, see [30]).

1. Create an execution path in the correct algorithm by randomly choosing a branch such that the branch coverage is maximised.

2. Generate input conditions based on the execution path of the correct algorithm.

3. Run the conditional input with a misconception algorithm. Choose branches similarly than in step 1.

[1]Some of these ideas were presented by Otto Seppälä the guidance meetings of this Master's Thesis.

4. Refine input conditions based on the execution path of the misconception algorithm.

5. Repeat steps 3 and 4 for other misconception algorithms.

6. If a contradiction is encountered, backtrack in a misconception algorithm or the correct algorithm and try another branch.

7. Choose fixed values of the restricted input such that IR4 and IR5 are fulfilled.

Step 1 is rather easy, as the elementary algorithms of VAS exercises are quite compact. The exploring algorithm must remember which branches have already been taken.

Step 1 tries to ensure IR3. Steps 2–6 tries to restrict the input space. Step 7 actually tries to fulfill IR4 and IR5. It still still involves random search and may fail to fulfill the requirements. It would be nice to check in steps 3 and 4 iteratively that IR4 and IR5 are fulfilled, but that would mean modifying an *abstract set of solution sequences*. It must be noted that a path in the algorithm might not be the same as the solution sequence; it should be studied exercise by exercise which is their connection.

3 Theoretical development

Algorithm 1 State-based similarity following [27, p. 246]

```
1:  procedure CLASSIFY-MISCONCEPTION(S_s, Algorithms)
2:      i_max ← 0; a_best ← ∅
3:      for a ∈ Algorithms do
4:          S_c ← a(S_s[0])
5:          i, j ← 0
6:          while i < |S_c| and j < |S_s| do
7:              while j ≤ |S_s| and S_c[i] ≠ S_s[j] do
8:                  j ← j + 1
9:              end while
10:             if j ≤ |S_s| then
11:                 i ← i + 1
12:             end if
13:         end while
14:         if i > i_max then
15:             a_best ← {a}; i_max ← i
16:         else if i = i_max then
17:             a_best ← a_best ∪ a
18:         end if
19:     end for
20:     return a_best
21: end procedure
```

3.1 Input generation by symbolic state exploration

This is continuation from Section **??**, and ideas of Khurshid, Păsăreanu, and Visser [12]. It might be possible to run the correct algorithm and the misconception algorithms all simultaneously, step by step, using symbolic execution, lazy initialisation and path exploration.

Algorithm 2 describes a draft for the systematic input generation. The algorithm has similar inputs than Algorithm **??**: *exercise* is the VAS exercise, *Algorithms* contains the correct algorithm and misconceived algorithms, and *length* is the length of the input.

Algorithm **??** utilises the idea of *path condition*:

Array *divergent* in line 1 of Algorithm 2 stores for each algorithm $A \in Algorithms$ a path condition where the execution sequence of A begins to diverge from the execution sequences of other algorithms. This is essential in trying to fulfill IR4 and IR5.

Step 2 indicates lazy initialisation. all the possible inputs are considered. When Algorithm 2 proceeds by exploring an execution path of an algorithm $A \in Algorithms$, it adds constraints to the input.

The *execution sequence* of an algorithm in VAS context depends on the execution path of an algorithm, but includes only the steps where the algorithm modifies a data structure. For an exchange-sorting and build-heap algorithms, this means swap of two elements of an array. For a graph algorithm this is a modification of the *visited* or *distance* arrays. For a hash table the steps should include both modification of the table and the array indices that are traversed with some probing function. Therefore the execution sequence is a candidate for the student's solution sequence in a VAS exercise.

Algorithm 2 Input generation by simultaneous path exploration (draft)

Input: $exercise, Algorithms, length$

1. Initialise *divergent*.
2. Begin with unconstrained input set of length l.
3. Run the correct algorithm and all the misconception algorithms step by step simultaneously:
4. When a branching point is encountered in the correct algorithm, choose the branch by following conditions.
   (a) Choose a branch that have not been explored earlier.
   (b) Choose a branch that makes the execution sequence of some misconception algorithm diverge from the others.
   (c) Choose randomly.
5. After a branch is chosen, record the step in the execution path history and constrain the input set according to the truth value of the branching condition.
6. If a contradiction is encountered, backtrack until a branching point with unxplored branches is reached. Choose an unexplored branch.
7. End search when the correct algorithm ends or all possible execution paths have been explored.
8. Generate fixed input by choosing randomly following the conditions of the symbolic input.

It might be useful to store the symbolic input of a successful path search after step 7. While systematic, the search might still be computationally expensive, and as many different inputs can be generated from the symbolic result, it is wise to reuse the result. Step 8 is actually an algorithm on its own and its design is a separate research question.

If steps 4 (a) is performed primarily, 4 (b) secondarily and 4 (c) tertiarily, the algorithm is greedy. This greediness might constrain the results.

The next step is trying this input generation algorithm with some concrete VAS exercise. What kind of conditions does the resulting symbolic input have based on the algorithm of the exercise, meaning sorting, binary heap, trees, hashing, and graph algorithms? Is it possible that the resulting conditions become too complex to be practical?

4 Empirical study

4.1 University course on data structures and algorithms

This thesis discusses teaching computer science (CS) at undergraduate university level. The CS major studies at entry level in Aalto University begins with basics of programming, mathematics and physics. The course named Data Structures and Algorithms (DSA) is on the second semester. [23] A similar course is also taken by students having a CS minor. Both courses have lectures, tutorial sessions and individual exercises, and have 200–300 students each year. The courses discuss basics of algorithm analysis and data structures, sorting, trees, hashing, and graph algorithms. [16] [17]

The teaching methods on the course are heavily computer-supported. Although the teaching staff on one DSA course is the lecturer and less than ten teaching assistants, and therefore average time the teacher can spend on discussion with one student is low, the course aims to have high quality teaching by utilising *automatic assessment*. In addition to a textbook, there are electronic learning material: quizzes, programming assignments and *visual algorithm simulation* exercises. All of these exercises are automatically assessed: the students works on the exercise, submits their solution and receives instant grading and feedback. [17] The quizzes might have teacher-written automatic feedback text depending on the choice of answers. The programming assignments are assessed by running a set of unit tests: in each test, the student-written program is run with some input and its output with comparison to expected result is shown to the student. Also the visual algorithm simulation exercises, where the students performs the steps of a given algorithm with given input, assess the correctness ratio of the solution and can show the model solution. The lecturer concentrates on designing the exercises and giving lectures. The teaching assistants give guidance to the students on the programming assignments on request. I have experience on being a teaching assistant on the course.

The difference between CS majors and minors in Aalto University is that CS majors learn Scala as their first language, and also recursion before taking the data structures and algorithms course, while the minors have Python. This thesis studies the visual algorithm simulation exercises and students' solutions to then on the data structures and algorithm course for computer science minors; the data is from years 2017–2018. However, earlier research has evidence that the studied misconceptions are generally similar with CS major and minors [27] [10].

4.2 Misconception-awareness in current JSAV-based exercises

This section validates the input generation of current JSAV-based VAS exercises against IR3.

4.3 Current learning analytics in A+ LMS on VAS exercises

Currently A+ LMS provides basic learning statistics on VAS exercises: the score distribution for each exercise, and the submission time, submission number and automatic score for each student and submission attempt. This data only helps the teacher analyse how much VAS

exercises the student do in comparison to other types of exercises on the course, and whether some VAS exercises are particularly easy or difficult.

In addition to the statistics recorded by A+, the LMS also stores JSON data for each VAS exercise submission. This data contains the input data of the exercise and the state of the data structure on each simulation step, which the JSAV library has recorded when the student has worked with the exercise and submitted it. Only this data could be detailed enough for misconception detection. However, it is not used currently, because its format is not human-readable in practise; a data visualiser similar to the model answer slideshow would be an ideal tool for this if only it existed.

4.3.1 Evaluating postfix expression

Algorithm 3 Evaluation of a postfix expression

```
1. read postfix expression token by token
2.   if the token is an operand, push it into the stack
3.   if the token is a binary operator,
3.1    pop the two top most operands from the stack
3.2    apply the binary operator with the two operands
3.3    push the result into the stack
4. finally, the value of the whole postfix expression remains in the stack
```

A postfix expression is a string consisting of operands (here: integers) and operators (here: binary operators addition + and multiplication $*$). The evaluation of a postfix expression is described in algorithm 3, which are the English instructions of the "Evaluating postfix expression" VAS exercise in the OpenDSA interactive textbook [22, AV/Development/postfixEvaluationPRO.json]. The exactly same file is in the git repository of the DSA Y course.

Based on this definition, a valid postfix expression with binary operators has x operands and $x - 1$ operators, as each operator takes two operands from the stack and returns one, effectively decreasing the size of the stack by one. Moreover, each time an operator is read from the input, there must be at least two operands in the stack. After the input is processed only one operand should remain in the stack, the result value. The length of the expression is odd, as $x + x - 1 = 2x - 1 > 0,\ x \in \mathbb{Z}$. An example of an expression that the VAS exercise generates is "`7 2 + 5 5 + 7 * * 7 2 + + 9 +`", where $*$ and + are the multiplication and addition operators, consecutively.

IR3 for Evaluating postfix expression are specifically:

1. The expression must always be valid.
2. The expression must always have both + and $*$ operators.

Algorithm 4 is the corresponding input generation part of the "Evaluating postfix expression" exercise implemented with JavaScript and JSAV [22, AV/Development/postfixEvaluationPRO.js]. Initially variable `arraySize` is set to 15 which is the length of the input. The input array `initialArray` is filled symbol by symbol.

Function `JSAV.utils.rand.numKey(1, 10)` returns an integer between $(1, 9)$ [24]. Function `JSAV.utils.rand.random()` is not documented, but the JSAV source code [18, utils.js] shows it is the builtin JavaScript `Math.random` function, which returns a floating-point number between $(0, 1($ [21].

The `if` condition on line 40 determines whether the symbol is an operand (an integer in range $(1, 9)$), or the operator `*` or `+`. In practise, both of the operators have an equal probability to be chosen each time (see line 47).

Algorithm 4 Input generation of Evaluating postfix expression [22, AV/Development/postfixEvaluationPRO.js]

```
 5 var arraySize = 15, // size needs to be odd
 6    initialArray = [],
34 // generate random postfix expression and put it in the array
35 var numbersInArray = 0,
36     randomVal,
37     i;
38 for (i = 0; i < arraySize; i++) {
39   // determine if the next character should be an operand or an
        operator.
40   if (numbersInArray < i - numbersInArray + 2 ||
41       JSAV.utils.rand.random() <
42       (Math.ceil(arraySize / 2) - numbersInArray) / (arraySize - i))
43   {
44     randomVal = JSAV.utils.rand.numKey(1, 10);
45     numbersInArray++;
46   } else {
47     randomVal = JSAV.utils.rand.random() < 0.5 ? "+" : "*";
48   }
49   initialArray[i] = randomVal;
50 }
```

The first part of the `if` condition is

$$\begin{aligned} numbersInArray &< i - numbersInArray + 2 \\ \Leftrightarrow numbersInArray &< (i+2)/2, \end{aligned}$$

which asserts the validity condition "x operands and $x-1$ operators". The input array has currently i symbols, of which should be $numbersInArray$ operators and $numbersInArray+1$ operands, yielding $numbersInArray = (i+1)/2$. The code has $i+2$ to make the division ceiling-rounding. If there are less than the required amount of operands, an operand will be generated next.

The second part of the condition on lines 41–42 is

$$r < \frac{\lceil arraySize/2 \rceil - numbersInArray}{arraySize - i}$$

where r is a random floating-point number from range $(0, 1($. This is the other condition to generate an operand if the first condition does not apply. This second condition becomes invalid when $numbersInArray$ has reached its maximum, $(arraySize-1)/2$, as then the numerator becomes negative and there is zero probability that $0 \leq r < 1$ is less than the right side. Table 1 shows the combined operation of the expressions which lead to generating an operand. Value 1 is the first subexpression and other values greater than zero is the second subexpression. The probability of generating an operand decreases as $numbersInArray$ increases, but it also increases when i increases.

The condition $numbersInArray < i - numbersInArray + 2$ ensures that there are enough operands in each generated postfix expression. As the algorithm proceeds, i increases by one every step and correspondingly $numbersInArray$ may increase by one. The zero area with $8 \leq numbersInArray \leq 14$ ensures that the postfix expression has enough operands. In

	numbersInArray								
i	0	1	2	3	4	5	6	7	8 . . . 14
0	1	0.47	0.40	0.33	0.27	0.20	0.13	0.07	0
1	1	1	0.43	0.36	0.29	0.21	0.14	0.07	0
2	1	1	0.46	0.38	0.31	0.23	0.15	0.08	0
3	1	1	1	0.42	0.33	0.25	0.17	0.08	0
4	1	1	1	0.45	0.36	0.27	0.18	0.09	0
5	1	1	1	1	0.40	0.30	0.20	0.10	0
6	1	1	1	1	0.44	0.33	0.22	0.11	0
7	1	1	1	1	1	0.38	0.25	0.13	0
8	1	1	1	1	1	0.43	0.29	0.14	0
9	1	1	1	1	1	1	0.33	0.17	0
10	1	1	1	1	1	1	0.40	0.20	0
11	1	1	1	1	1	1	1	0.25	0
12	1	1	1	1	1	1	1	0.33	0
13	1	1	1	1	1	1	1	1	0
14	1	1	1	1	1	1	1	1	0

Table 1: Probability of generating an operand in the `if` branch of algorithm 4 with given values of variables and $arraySize = 15$.

an extreme case, the algorithm first generates maximum number of operands. Then $i = numbersInArray = 8$. According to Table 1, the probability of generating an operand is now 0. $numbersInArray$ will not increase further, and thus the rest of the input is operands. Therefore algorithm 4 always creates a valid postfix expression.

Other properties of Algorithm 4 that could violate IR3 are (a) generating only one type of operand and (b) generating only one type of operator. The probability for case (a), is

$$\frac{9 \times (1/9)^8}{9^8} \approx 5 \times 10^{-15},$$

which is practically "never". The probability for case (b) is $0.5^7 \approx 0.0078$, or $1 : 128$. If 200 students take the course each year and each of them does the "Evaluating postfix expression" exercise approximately once, 1–2 of them will receive a postfix expression with only +'s or $*$'s. This case fails to reveal the hypothetical misconception that the input is not processed linearly in one pass, as one can execute the individual operators in the input in an arbitrary way. Therefore this exercise fails IR3 at probability of $1/128$. □

The input could be corrected simply by counting each operator type in the generated postfix expression and changing some operator to the missing type at random location. Adding subtraction and division operators would also help, but then the expression should be validated against division by zero, and by default, the values in the stack become floating-points types.

4.3.2 Quicksort

The Quicksort JSAV exercise presents the student a reference code resembling C++ and Java languages, as shown in Algorithm 5. Regarding IR3, branches on lines 16 and 17 must be both taken and not taken at some part of the recursion. This happens naturally, as these are the end conditions of the recursion. Therefore the actual branch variance is in the `partition` function. This function is given a `pivot` value, and it will reorganise the range [`left`, `right`] in array `A` such that the elements less and equal than the pivot are first and elements greater than the pivot are after that.

Algorithm 5 Reference code shown in the Quicksort JSAV exercise [22, SourceCode/Processing/Sorting/Quicksort.pde]

```
 1 int partition(Comparable[] A, int left, int right, Comparable pivot) {
 2   while (left <= right) { // Move bounds inward until they meet
 3     while (A[left].compareTo(pivot) < 0) left++;
 4     while ((right >= left) && (A[right].compareTo(pivot) >= 0))
         right--;
 5     if (right > left) swap(A, left, right); // Swap out-of-place values
 6   }
 7   return left;            // Return first position in right partition
 8 }
 9
10 void quicksort(Comparable[] A, int i, int j) { // Quicksort
11   int pivotindex = findpivot(A, i, j);  // Pick a pivot
12   swap(A, pivotindex, j);               // Stick pivot at end
13   // k will be the first position in the right subarray
14   int k = partition(A, i, j-1, A[j]);
15   swap(A, k, j);                        // Put pivot in place
16   if ((k-i) > 1) quicksort(A, i, k-1);  // Sort left partition
17   if ((j-k) > 1) quicksort(A, k+1, j);  // Sort right partition
18 }
```

Algorithm 5 is the JavaScript source code of the exercise, which shows both the pivot function and the input generation. Variable `arraySize` on line 53 is the input size, which is 10. The pivot function is median of values in the first, middle and last indices of the range.

The pivot function decides how the array is partitioned recursively. However, at each recursion step, the array is always partitioned into two. The choice of pivot is automatic in the exercise, and therefore its execution is not needed to be analysed in detail.

The most important issue regarding IR3 is therefore whether the student has to perform at least one `swap` operation in the partition function, precisely on line 5 in Algorithm 5. The only case where no such swaps are needed is the one where the input is already sorted. Lines 4 and 53 in Algorithm 6 indicates the input is 10 integers which are independently and randomly chosen from range $(10, 125($. Therefore the problem of IR3 regarding Quicksort VAS reduces to estimating the probability of creating a random input which is already sorted, and deciding whether if the probability is significant in practise.

Denote the number of elements in the array by k and the size of the sampling space by s. It is assumed that $s > k$. The actual values for this exercise are $k = 10$ and $s = 124-10+1 = 115$. Note that the integer range $(10, 125$ can be bijectively mapped onto natural numbers $(1, 115)$. There can be several duplicate values, or $1 \dots k$ unique values. Denote the input sequence by $1 \leq (a_i) \leq s$ where $1 \leq i \leq k$. The condition for "all sorted cases" is then $a_i \leq a_{i+1}$ when $1 \leq i \leq k-1$.

For $k = 1$, there are $f(s, k = 1) = s$ cases.

For $k = 2$, enumerating cases $s = 2, 3, 4$ reveals the following exact cases. Here `(1 2)` denotes an input sequence (ordered tuple) with integer value 1 first and integer value 2 after that.

```
s = 2, k = 2
------------
(1 1) (1 2)
(2 3)                    Total 2 + 1 = 3
```

Algorithm 6 Quicksort pivot function and input generation [22, AV/Development/quicksort2PRO.js]

```
 4   var arraySize = PARAMS.size ? parseInt(PARAMS.size, 10): 10,

20 var pivotFunction = {
21   last: function (left, right) { return right; },
22   middle: function (left, right) { return Math.floor(
         (right + left) / 2); },
23   medianof3: function (left, right, arr) {
24     var mid = this.middle(left, right);
25     var median = [arr.value(left), arr.value(mid), arr.value(right)]
         .sort(function (a, b) { return a - b; })[1];
26     if (arr.value(right) === median) {
27       return right;
28     } else if (arr.value(mid) === median) {
29       return mid;
30     }
31     return left;
32   }
33 };

53 initialArray = JSAV.utils.rand.numKeys(10, 125, arraySize);
```

```
s = 3, k = 2
------------
(1 1) (1 2) (1 3)
(2 2) (2 3)
(3 3)                    Total 3 + 2 + 1 = 6

s = 4, k = 2
------------
(1 1) (1 2) (1 3) (1 4)
(2 2) (2 3) (2 4)
(3 3) (3 4)
(4 4)                    Total 4 + 3 + 2 + 1 = 10
```

Thus it is obvious that for $k = 2$, the number of elementary cases are

$$f(s, k = 2) = \sum_{i=1}^{k} i = \frac{(s+1)s}{2}.$$

The structure of case $k = 3$ bears similarity to case $k = 2$:

```
s = 3, k = 3
------------
(1 1 1) (1 1 2) (1 1 3)
(1 2 2) (1 2 3)
(1 3 3)

(2 2 2) (2 2 3)
(2 3 3)
```

```
(3 3 3)

Total 6 + 3 + 1 = 10

s = 4, k = 3
------------
(1 1 1) (1 1 2) (1 1 3) (1 1 4)
(1 2 2) (1 2 3) (1 2 4)
(1 3 3) (1 3 4)
(1 4 4)

(2 2 2) (2 2 3) (2 2 4)
(2 3 3) (2 3 4)
(2 4 4)

(3 3 3) (3 3 4)
(3 4 4)

(4 4 4)

Total 10 + 6 + 3 + 1 = 20

s = 5, k = 3
------------
(1 1 1) (1 1 2) (1 1 3) (1 1 4) (1 1 5)
(1 2 2) (1 2 3) (1 2 4) (1 2 5)
(1 3 3) (1 3 4) (1 3 5)
(1 4 4) (1 4 5)
(1 5 5)

(2 2 2) (2 2 3) (2 2 4) (2 2 5)
(2 3 3) (2 3 4) (2 3 5)
(2 4 4) (2 4 5)
(2 5 5)

(3 3 3) (3 3 4) (3 3 5)
(3 4 4) (3 4 5)
(3 5 5)

(4 4 4) (4 4 5)
(4 5 5)

(5 5 5)

Total 15 + 10 + 6 + 3 + 1 = 35
```

By further study, it is obvious that case $k = 3$ is a recursive sum of cases $k = 2$:

$$f(s, k = 3) = \sum_{i=1}^{s} f(i, k = 2) = \sum_{i=1}^{s} \sum_{j=1}^{i} j$$

More generally, by further enumerating the cases, it is obvious that the number of cases is a k-dimensional sum which can be defined recursively:

$$f(s,k) = \begin{cases} s & k = 1 \\ \sum_{i=0}^{s-1} f(i, k-1) & k > 1 \end{cases} \tag{1}$$

This recursive definition has connection to binomial coefficients: cases for different values of k can be read from Pascal's triangle as Figure 2 shows.

k \ s	1	2	3	4	5
1	1	2	3	4	5
2		3	6	10	15
3			10	20	35
4				35	70
5					126
k					

(a)

n \ k	0	1	2	3	4	5	6	7	8	9	10	11	k
0	1	1	1	1	1	1	1	1	1	1	1	1	
1	1	2	3	4	5	6	7	8	9	10	11		
2	1	3	6	10	15	21	28	36	45	55			
3	1	4	10	20	35	56	84	120	165				
4	1	5	15	35	70	126	210	330					
5	1	6	21	56	126	252	462						
6	1	7	28	84	210	462							
7	1	8	36	120	330								
8	1	9	45	165									
9	1	10	55										
10	1	11											
11	1												
n													

(b)

Figure 2: (a) $f(s,k)$ computed by enumeration; (b) Binomial coefficients $\binom{n}{k}$ in Pascal's triangle.

Thus it is claimed that $f(s,k)$ has a closed, nonrecursive form:

$$f(s,k) = \binom{s+k-1}{k} \tag{2}$$

Indeed, Knuth [13, p. 56] defines a property of binomial coefficients which resembles equation (1):

$$\sum_{0 \le k \le n} \binom{k}{m} = \binom{n+1}{m+1} \tag{3}$$

This can be proven by induction. Assumption. f is a binomial coefficient: $f(s,k) = \binom{x}{y}$. Base case. $k = 1$. Begin with equation (1).

$$f(s,k) = s = \binom{s}{y},$$

meaning $x = s$ and $y = k = 1$. Thus the base case holds.

Induction step. Assume the claim holds with at most some k. $f(s,k) = \binom{x}{y}$ where $x = s$. Now begin with

$$f(s,k+1) = \sum_{i=0}^{s-1} f(i,k) = \sum_{i=0}^{s-1} \binom{i}{y}\Bigg|_{eq.(3)} = \binom{(s-1)+1}{y+1} = \binom{s}{y+1}$$

Thus $f(s,k) = \binom{x}{y}$ and $f(s,k+1) = \binom{x}{y+1}$. Clearly $k = y$ and the induction proof is completed successfully.□

Therefore, using equation (2), it is possible to obtain the precise probability for having an already sorted input with k elements chosen independently from sample space having s elements. The probability is

$$\frac{f(s,k)}{s^k} = \frac{\binom{s+k-1}{k}}{s^k} = \frac{\frac{(s+k-1)!}{k!(s+k-1-k)!}}{s^k} = \frac{(s+k-1)!}{k!(s+k-1-k)!s^k}$$
$$= \left[\prod_{i=1}^{k} \frac{s+k-1}{s}\right] \frac{1}{k!}$$

The probability for $s = 115$ and $k = 10$ is $\approx 4.03 \cdot 10^{-7}$, roughly one in million. If the DSA course is taken by 300 each students and each of them try the exercise 3 times on average, the probability that on one year none of them ever sees an input which is already sorted is

$$\left(1 - 4.03 \cdot 10^{-7}\right)^{300 \cdot 3} \approx 0.9996$$

This is a tolerable failure probability. Therefore the exercise satisfies IR3.□

5 Player for exercise recordings

This section studies the research question RQ5: *Is it possible to construct a "player application" for the current submissions of the JSAV exercises?*

5.1 Reproducibility of the current exercise recordings

5.1.1 Infix to postfix

The recording of this exercise consists of a JSON object which as a list containing elements that display the current state of the output array containing the result, a postfix expression. Figure 3 shows an example. Field `ind` describes an array with 11 elements. The three first elements in the array are `8`, `c`, and `*`, while the rest of the array is empty. Data of fields `style` and `classes` remain invariant for every step.

```
{
  "ind": [{ "v": "8" }, { "v": "c" }, { "v": "*" }, { "v": "" },
          { "v": ""  }, { "v": "" },  { "v": ""  }, { "v": "" },
          { "v": ""  }, { "v": "" },  { "v": ""  }],
  "style": "height: 47px; width: 508px;",
  "classes": ["jsavcenter", "jsavautoresize"]
}
```

Figure 3: Example step of a JSON recording of the Infix to postfix exercise.

Table 2 shows all the essential data of the example recording. The output array is filled character by character from left to right. This particular submission was automatically graded as correct; the score data is also available in A+, and it can be combined with the recording. Notice that for some adjacent steps, such as 6–8 and 15–16, the output array remains unchanged. These likely indicate steps where the user marks a right parenthesis or a left parenthesis in the input as processed.

The recording of this exercise fails RR1. The initial input is not given. Simply validating whether the output is a valid postfix expression is not sufficient.

First, although this exercise has binary operators that are commutative, that is, $a+b = b+a$ and $a * b = b * a$, it is important that the student can run the algorithm correctly such that

Step(s)	Postfix array										
1–2											
3–4	8										
5	8	c									
6–8	8	c	*								
9–11	8	c	*	e							
12–13	8	c	*	e	1						
14	8	c	*	e	1	e					
15–16	8	c	*	e	1	e	+				
17	8	c	*	e	1	e	+	*			
18–19	8	c	*	e	1	e	+	*	+		
20	8	c	*	e	1	e	+	*	+	9	
21–22	8	c	*	e	1	e	+	*	+	9	+

Table 2: All steps of an example recording of the Infix to postfix exercise.

the operands are processed in the correct way. For example, infix expression "`a + b`" has the correct postfix expression form "`a b +`", but also the infix expression "`b + a`" can be converted incorrectly as "`a b +`".

Second, regarding operator precedence, one postfix expression can also have both a correct infix form and incorrect infix forms that could be misinterpreted as the correct result. For example, a valid postfix expression "`a b c + *`" has the correct infix input "`(c + b) * a`", but it is possible that someone constructs the same end result from input "`a * c + b`"; the user interface of the exercise allows this.

The exercise also fails RR2, as it is not indicated which particular left and right parentheses in the input are processed at which steps. Therefore the submissions to the *Infix to postfix* exercise are not currently reproducible.□

5.1.2 Quicksort

Discussing the graphical user interface (GUI) of the Quicksort JSAV exercise is important for two reasons. First, the GUI color-codes array elements as the exercise proceeds to guide the student, and this color coding is also present in the exercise recording. Second, the GUI constrains user's actions, which also guides the interpretation of the exercise recordings. The GUI is shown in figure 4. The exercise itself chooses a pivot and swaps it to the end of the current range; this corresponds to lines 11–12 in algorithm 5. Pivot value is 91 in figure 4 and it has already been moved to the last index.

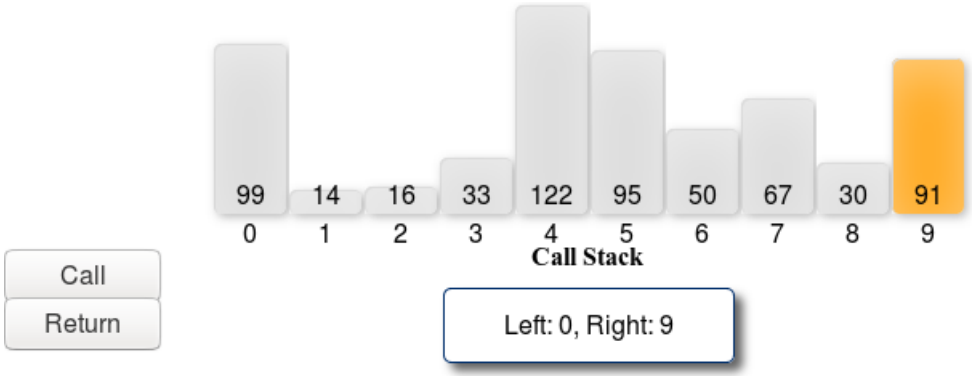


Figure 4: GUI of the Quicksort JSAV exercise in its initial state.

The student must perform the PARTITION procedure (lines 1–8 and 14 in algorithm 5), swap the pivot to correct location (line 15 in 5), and finally call the QUICKSORT procedure recursively for the left and right parts (lines 16–17 in 5). Making a call to QUICKSORT is done by clicking the first element in the array including the range, then clicking the last element in the array including the range, and finally clicking a *Call* button.

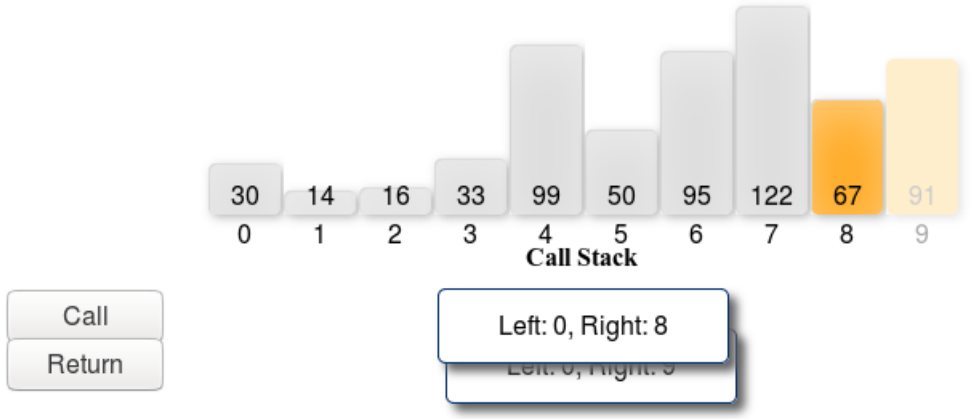


Figure 5: GUI of the Quicksort JSAV exercise after calling Quicksort(0, 8).

After clicking the *Call* button, the elements of the array that were not on the call range become light-colored, as the array index 9 in figure 5. There QUICKSORT was just called recursively for range [0,8]: array elements at indices 0 and 8 were clicked and the *Call* button was clicked. Then the current pivot was chosen automatically and moved to index 8. Now the student must do the PARTITION procedure for range [0,7]. Note that the bottom of the GUI displays the call stack for QUICKSORT: there is the first range [0, 9], and the top of it, the range [0, 8]. Correspondingly, there is a *Return* button which the student must click to indicate returning from a recursive call of QUICKSORT.

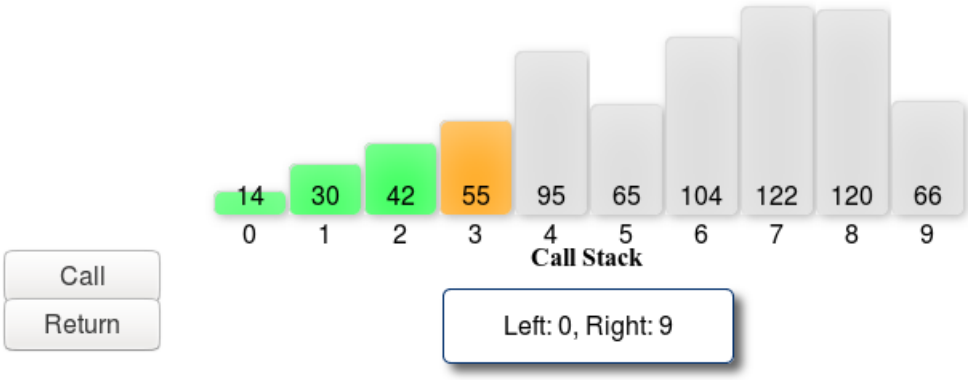


Figure 6: GUI of the Quicksort JSAV exercise after returning from Quicksort(0, 2).

The exercise only allows making a recursive call for a range that does not extend the range of the current call: if the current range is $[l_0, r_0]$, then for the new range $[l_1, r_1]$ it must be $l_0 \leq l_1$ and $r_1 \leq r_0$. If the student chooses $l_1 > r_1$, all the array elements in the GUI become light-colored indicating that the student cannot proceed, which likely leads the student clicking *Return*, *Undo*, or *Reset*. This is shown in Figure 6.

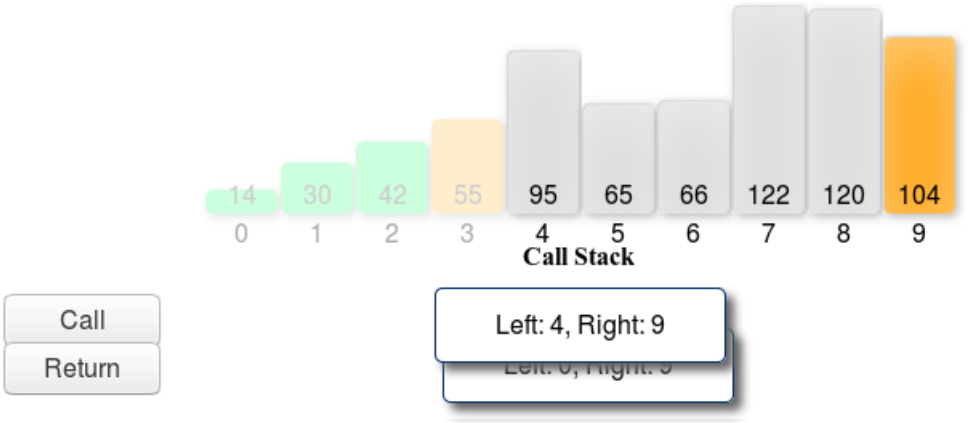


Figure 7: GUI of the Quicksort JSAV exercise after calling Quicksort(4,9).

Figure 7 shows how the parts of the array which are known to be processed remain green in the exercise even at the further steps. When another call to QUICKSORT is made, the green becomes light green similar to the hint that an upper-level pivot is shown as light orange. Finally, when the student clicks the *Return* button when there is only QUICKSORT(0, 9) in the call stack, all elements of the array become green.

Similar to the Infix-postfix exercise, the Quicksort exercise records the contents of an array for each step, as shown in Figure 8. Field `ind` contains an array of size 10, each index holding an integer value, and one of the indices is highlighted as the current pivot value (`"cls": ["pivot"]`). The `style` and `classes` fields remain invariant.

Moreover, at each step, each of the array indices can have several "class" values defined by the `cls` field. The observed classes are described in Table 3.

The colors of classes are the same as in the actual user interface of the Quicksort VAS exercise. Only the jsavarrow class is actually represented as a little black arrow pointing to one array index.

A full example of the actual data recorded by the Quicksort exercise is shown in Table 4. This recording features a correct submission. Step 1 is the situation after Quicksort has been

```
{
  "ind": [{ "v": 16 }, { "v": 15 }, { "v": 14 }, { "v": 53 },
    { "v": 50 }, { "v": 54, "cls": ["pivot"] }, { "v": 98 },
    { "v": 56 }, { "v": 88 }, { "v": 71 }],
  "style": "width: 471px; top: 40px;",
  "classes": ["jsavcenter", "jsavautoresize"]
},
```

Figure 8: Example step of a JSON recording of the Quicksort exercise.

Class	Description
(none)	Unprocessed region
pivot	current pivot
jsavarrow	the leftmost index of an area to be processed when beginning a recursive call to Quicksort
inactive, pivot	pivot of earlier recursive step, held in call stack
inactive	right half of a recursive step waiting for Quicksort call
green	a Quicksort call completed; just finished region
inactive, green	earlier, finished region

Table 3: Array cell classes in the recording of a Quicksort exercise

called for the whole array, the pivot element has been chosen automatically, and the pivot has been swapped in the end of the array. Reminding algorithm 6, the selection of the pivot is median of three values: the first, the middle, and the last values. Because the initial, random input has not any constraints regarding the relative order of the elements, there are $3! = 6$ different inputs with which choosing the pivot and swapping it to the last index results to identical array. However, because the pivot is chosen automatically, we can freely choose any of these inputs. The easiest choice is that the last element in the input already contains the median, and thus not any swap is required. Therefore Quicksort exercise satisfies RR1.

Swap operations can be deduced inambiguously by comparing two adjacent steps. This is exactly how the swap boldfacing is done in Table 4.

Considering the GUI, the following is an interpretation of recording in Table 4.

Step 1 is the initial step.

Steps 2–3 run Partition for range [0,8], because this range is active and none of the swaps involve a pivot.

Step 4 swaps the pivot to the correct position.

Step 5 seems to be idle. It is assumed that the user has clicked the *Call* button.

Step 6 indicates index 0 will be the beginning index for the recursive call.

Step 7 has pivot at index 4. Combining this information with step 6, it is logical that Quicksort(0,4) was called. Elements at indices 2 and 4 were swapped automatically by the exercise.

Step 8 involves swapping a pivot. This means that no swapping was done in the call of Partition(0,3).

Step 9 is idle; presumably the user clicks the *Call* button.

Step 10 shows the beginning of the range for recursive Quicksort call.

Step 11 is similar to step 7: unambiguously, Quicksort(1,4) was called, elements at indices 2 and 4 were swapped, and the current pivot is now at index 4.

Step 12 has a swap involving the current pivot. The logic is similar to the step 8; Parti-

Step(s)	Array									
	0	1	2	3	4	5	6	7	8	9
1	16	98	14	71	50	53	15	56	88	54
2	16	**15**	14	71	50	53	**98**	56	88	54
3	16	15	14	**53**	50	**71**	98	56	88	54
4–5	16	15	14	53	50	**54**	98	56	88	**71**
6	16	15	14	53	50	54	98	56	88	71
7	16	15	**50**	53	**14**	54	98	56	88	71
8	**14**	15	50	53	**16**	54	98	56	88	71
9	14	15	50	53	16	54	98	56	88	71
10	14	15	50	53	16	54	98	56	88	71
11	14	15	**16**	53	**50**	54	98	56	88	71
12	14	15	16	**50**	**53**	54	98	56	88	71
13	14	15	16	50	53	54	98	56	88	71
14	14	15	16	50	53	54	98	56	88	71
15	14	**16**	**15**	50	53	54	98	56	88	71
16	14	**15**	**16**	50	53	54	98	56	88	71
17	14	15	16	50	53	54	98	56	88	71
18	14	15	16	50	53	54	98	56	88	71
19-20	14	15	16	50	53	54	98	56	88	71
21	14	15	16	50	53	54	98	56	88	71
22	14	15	16	50	53	54	98	**71**	88	**56**
23	14	15	16	50	53	54	**56**	71	88	**98**
24	14	15	16	50	53	54	56	71	88	98
25	14	15	16	50	53	54	56	71	88	98
26	14	15	16	50	53	54	56	71	**88**	**98**
27	14	15	16	50	53	54	56	71	88	98
28	14	15	16	50	53	54	56	71	88	98
29	14	15	16	50	53	54	56	71	88	98
30	14	15	16	50	53	54	56	71	88	98
31	14	15	16	50	53	54	56	71	88	98

Table 4: All steps of an example recording of the Quicksort exercise. Color codes are in Table 3. **Boldface** text highlights swapped array elements.

TION(1,2) did nothing.

Step 13 features implicitly clicking the CALL button.

Step 14 indicates the leftmost index of the next recursive call.

Step 15 is after calling QUICKSORT(1,2) and swapping pivot to the right.

Step 16 shows that PARTITION(6,6) did nothing and the pivot was swapped to its correct position.

Step 17 features clicking the RETURN button. QUICKSORT(1,2) ends.

Step 18 features clicking the RETURN button. QUICKSORT(1,4) ends.

Step 19 features clicking the RETURN button. QUICKSORT(0,4) ends.

Step 20 features clicking the CALL button.

Step 21 indicates that index 6 is the beginning of the range for the next call to QUICKSORT.

Step 22 indicates that QUICKSORT(6,9) was called, and elements at indices 7 and 9 were swapped to have pivot at index 9.

Step 23 indicates that PARTITION(6,8) did nothing and the pivot was swapped into another position.

Step 24 features clicking the CALL button.

Step 25 shows that QUICKSORT(7,9) was called and elements at indices 8 and 9 were swapped to have pivot at index 9.

Step 26 indicates that pivot is already at index 9 and no swap was required.

Step 27 shows moving the pivot to its correct position.

Step 28 features clicking the RETURN button. QUICKSORT(7,9) ends.

Step 29 features clicking the RETURN button. QUICKSORT(6,9) ends.

Step 30 features clicking the RETURN button. QUICKSORT(0,9) ends.

These steps can be described together by Algorithm 7. Therefore the Quicksort exercise fulfills RR1 and RR2 and is thus reproducible.□

Algorithm 7 Reconstruction of a Quicksort-JSAV recording

```
The input is a recording R which is a sequence of N steps.
Each step contains an array of size M. Each element of the array
has fields: v, an integer value, and cls, list of strings.
Initial input is R[0].array.
For i = 1 ... N-1:
  Compare fields v of R[i-1].array and R[i].array index by index.
    If two values with the same index are different, mark them as swapped.
  If R[i].array.cls contains a range of elements with "green" class, this
    the range for which a call of Quicksort has returned. Pop this
    call from the call stack.
  If R[i].array.cls contains an element with "jsavarray" class, memorise
    this as the left boundary of a Quicksort call.
  If a swap happened:
    If the swap involves an element with class "pivot":
      If a left boundary of a Quicksort call was updated at i-1:
        Now the pivot is the right boundary of a Quicksort call.
        Push this call to the call stack.
      Else:
        A Partition prodecure has finished.
    Else:
      The swap happened in a Partition procedure.
```

5.2 Red-black tree coloring

The remarks on this exercise rise from the thesis worker's discussion with a student taking the Autumn 2019 instance of the DSA course. The student expressed inability to understand the automatic grading of the exercise. The thesis worker examined the student's submissions to the exercise. The results show that the submissions are reconstructible, and that there is one particular misconception.

This exercise presents the student a red-black tree with 25 nodes, each having a random integer value from range (25, 99), excluding duplicates [22, AV/Development/redBlackTreeColoring.js]. Initially all the nodes are red. The student can change the color of a node by clicking it. The student must choose either black or red color for each node according to the following four coloring rules [22, AV/Development/redBlackTreeColoring.json].

1. A node is either red or black.
2. The root is black.

3. A red node's child nodes are black.

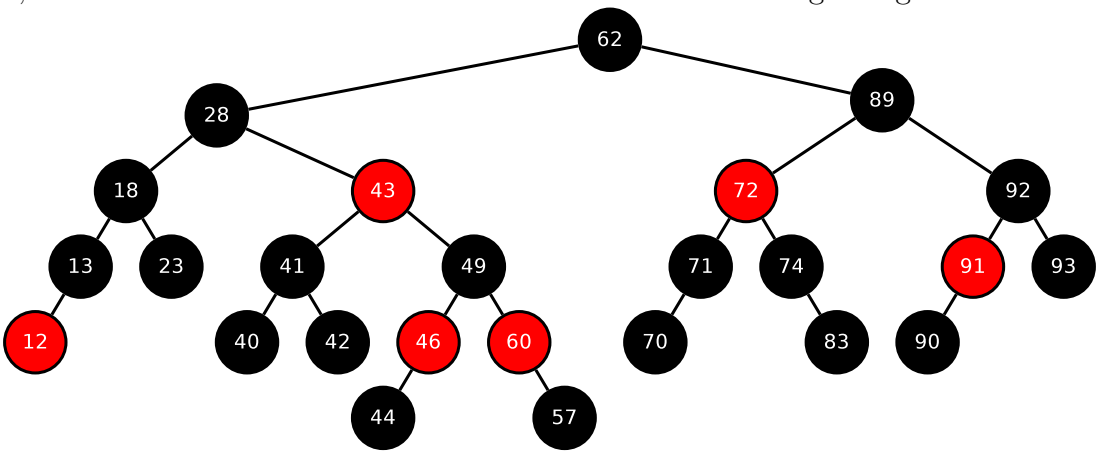
4. Each path from the root to an empty subtree contains the same number of black nodes.

The recording of the exercise consists of two steps. The first step describes the input: a Red-black tree with 25 nodes, each having an integer key, and all nodes colored red. The second step is the student's end result. Figure 9 shows a part of a the student's result. The top node in the subtree has value 43 (`"v": 43`) and color red (`"cls": ["rednode", ...]`). Its left child has value 41 and color black. The left-left grandchild has value 40 and color black. The left-right grandchild has value 42 and color black. Thus the subtrees contain subtrees recursively in the JSON data. The exercise recording also has positions of each node in the visual layout as the JSAV library has computed them.

```
"v": 43,
"cls": ["rednode", "jsavredblacknode"],
"css": "left: 223.25px; top: 104px;",
"left": {
  "v": 41,
  "cls": ["blacknode", "jsavredblacknode"],
  "css": "left: 157px; top: 156px;",
  "left": {
    "v": 40,
    "cls": ["blacknode", "jsavredblacknode"],
    "css": "left: 130.5px; top: 208px;"
  },
  "right": {
    "v": 42,
    "cls": ["blacknode", "jsavredblacknode"],
    "css": "left: 183.5px; top: 208px;"
  }
```

Figure 9: A subtree in the exercise recording of the "Red-black tree coloring" exercise.

Red-black tree coloring satisfies RR1, because the first step describes the input. Regarding RR2, the order in which the nodes are colored does not affect grading.



6 Other Stuff

6.1 The requirements and computational complexity of constained input generation

Research question. Assume constrained input generation of a VAS exercise described in [15]. If there is a VAS exercise r which has n elements in its input data, how many times x the random input data must be generated before the sequence of the correct solution is different from those produced by known misconception algorithms $M_r = \{m_{r,1}, \ \ldots \ m_{r,|M_r|}\}$, and also all sequences of the misconception algorithms differ from each other? This is a probability distribution of random variable X with parameters r, n, and M_r. The analytic solution of X is likely too hard to be solvable for all exercises and their misconceptions due to the underlying combinatorics.

Empirical testing is an alternative to the analytic study of optimal values for (x, n), given r and M_r. It must be noted that $n_{\min} \leq n \leq n_{\max}$: too short an input data will result in too easy an exercise for the student, and it will not reveal all misconceptions. On the other hand, too large n results in a very labourious and error-prone exercise for the student, and the student might become frustrated and skip the exercise.

6.1.1 Implementing the mutating misconception matching

Onko Oton väärinymmärrysmutaatiotutkimuksessa [26] mutantin haku tehty siten, että aina, kun koodi muuttuu, se käännetään uudelleen? Mutaatio voi tapahtua useamman kerran samassa koodipaikassa, joten tällä perusteella ei. Tutkimuksessa oli kuitenkin väite, että mutaatiohaun toteuttamiseen riittäisi yksinkertainen jäsentäjä. Tietty koodiin voi lisätä laskureita: "ensimmäisellä suorituskerralla tämän if-lauseen ehto on tämä, toisella suorituskerralla tämä, kolmannella tämä".

Laskennallisesti tehokas ohjelma, joka etsisi väärinymmärrystä opiskelijan tehtäväaskeleista metamutantin avulla, olisi esimerkiksi C++:lla toteutettu tulkki, jossa metamutantin kieli olisi rajattua C++:aa tai Javaa. Python olisi vähemmän sopiva metamutantin kieleksi, koska se ei ole vahvasti tyypitetty ja siten tulkki joutuu ajon aikana tulkitsemaan tilanteita.

6.2 Planning the solutions

There are two approaches to creating the automatic misconception detector. One is writing an expert system which detects a systematic misconception with a tailored algorithm or set of rules. This might be sensible because each systematic misconception can be described as an algorithm. This approach requires detailed analysis of the students' algorithm simulation answers, understanding which kind of modified algorithm they are applying, and then writing a detection algorithm for this modified algorithm. A variant of this is to run the initial data of an exercise with each of the misconceived algorithms and compare the similarity of their results to the student's end state of the exercise.

Another approach is to classify the misconceptions with machine learning. This approach requires still analysis of the students' answers and manually labeling the answers to several misconception classes to be able to use the answers as training data. This approach likely requires feature engineering and suitable algorithmic methods. Because the simulations are time series, the dynamic time warping algorithm might be useful tool for building a k nearest neighbor classifier for the students' answers.

The problem with the machine learning approach is that each student receives a random instance of the problem; for example, a sorting exercise begins with an array of 10 integers each selected randomly and independently between values 1 and 99. This randomness requires

some kind of data normalization and feature engineering to make all the students' answers comparable. One practical or theoretical end result might be that the randomness of the initial data for each exercise instance for each student must be controlled. For sorting exercises there could be, for example, ten permutations of an array, each of which gives a unique sequence of two-element swaps with given algorithm to reach the sorted end state. Moreover, the actual integer values of the permuted elements could be randomized as long as their relative order is preserved. (Example: permutation (4 2 1 3) could map to (23 8 7 19) or (10 5 2 6).

A sidenote on machine learning: deep neural networks would likely not work in this problem, because the size of training data per exercise is only some thousands of submissions, and each of these submission samples have only tens of steps.

Using unsupervised learning for automatic clustering of the submissions would be interesting, but it is likely that it will fail even if carefully constructed feature functions are used. That is, the brute-force human analysis of the students' answers is the key to understand the data, and clustering algorithms might at best provide some additional insight for the similarities between misconception classes.

There are in order of ten types of visual algorithm simulation exercises. The scope of this thesis might limit the study to only one exercise and constructing the software to classify them. Beginning with first and easiest algorithm simulation exercises, selection sort and insertion sort, is likely practical.

7 Player software

7.1 Design goals

The ultimate goal in facilitating teacher's work should be the design and development of a high-quality software application for the teacher. While this is out of the scope of this thesis, this section includes a draft the requirements and a graphical user interface, as they might be beneficial for the future work.

Regarding Section **??**, the teacher's use scenario is:

1. Review an incorrect exercise submission that did not match to a known misconception.
2. Write candidate for a misconceived algorithm that explains the submission.
3. Inspect how the candidate algorithm matches to the submission.
4. Edit the candidate, if needed.
5. In one case, label the submission as reviewed but inexplainable.
6. In another case, save the new misconceived algorithm with written, corrective feedback.

This use scenario translates directly into functional requirements. It seems beneficial if there was a single application that could assist the teacher with all the subtasks listed. Figure 10 features a draft of the graphical user interface for such an application.

The GUI in Figure 10 is shown as a traditional, windowed application in a desktop computer operating system, such as Microsoft Windows, macOS, or a Linux distribution. The application could also be a web application which is integrated into the LMS providing the VAS exercises; this is a separate engineering aspect. The *Exercise* drop-down list allows the teacher to select an exercise; currently the Build-heap exercise is selected. As there are hundreds of exercise submissions for a single exercise, the *Score range* selector might help selecting a subset of exercises that have scores within predefined range; the figure refers to 70–79% correct score.

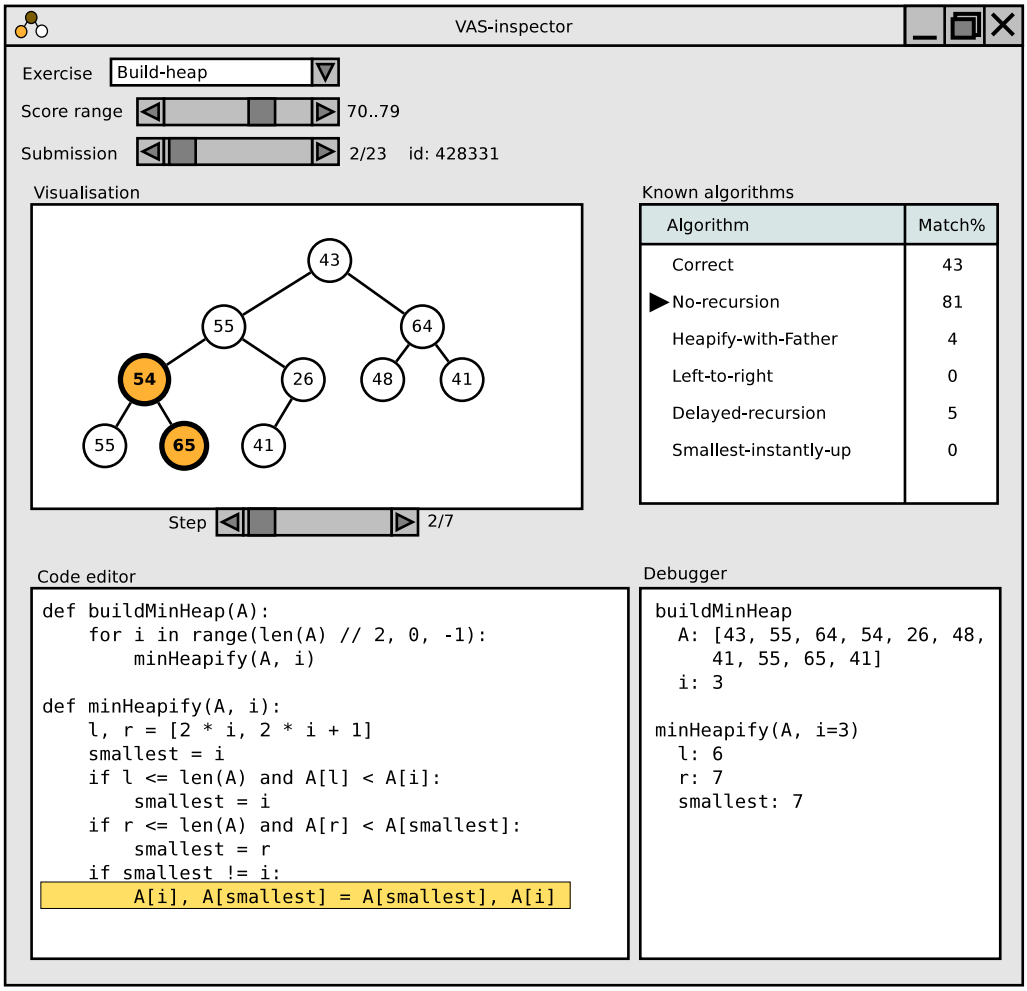


Figure 10: A proposal for the GUI of VAS-inspector

The *Submission* selector chooses the individual exercise submission. The *id* number refers to an unique exercise submission identifier in the LMS.

Figure 10 features a *Visualisation* view for the exercise submission. The teacher can see the exactly same visualisation and input that the student was working on, and the teacher can view student's actions step by step forward and backward as in a slideshow. As an example, there is a binary heap with elements 54 and 65 highlighted to indicate a swap operation.

To the right of the visual view, there is a list of *Known algorithms* for the exercise: the correct algorithm and misconceived algorithms, such as "No-recursion". The application shows the relative match of each algorithm for the current submission.

The bottom of the window in Figure 10 has a *Code editor*. When the user selects an algorithm from the list of *Known algorithms*, its program code is shown in the code editor. The code editor also works as a graphical debugger: the user can run the selected algorithm step by step and inspect its execution relative to the student's steps. The *Debugger* view shows current state of the program as values of variables. The highlighted line in the Code editor indicates the next line of program execution.

The idea of this section was to provide ideas for future work. Careful requirements engineering and user-centered design work should be executed if this kind of application was decided to be worth of developing. For example, one must notice that line by line execution of algorithms is a operationally more fine-grained inspection than, for example, viewing the student's swap actions in a binary heap VAS exercise. Moreover, the teacher might benefit from a side-by-side view of the student's simulation steps and steps performed by a known algorithm for the

particular exercise. Reconstructing the student's view is just one way of reviewing student's actions; using other software visualisation techniques for summarising an execution path might be beneficial.

7.2 Forming hypotheses for Build-heap misconceptions

Additional phenomena, if not straightforward misconceptions, seem to exist. Some sequences, named here as *Imitate-swaps*, have a sequence of swaps that represent a correct execution of the Build-heap algorithm, but not with the given input. These submissions have several swaps where a lower priority element is swapped upwards, or several times the wrong child is swapped with its parent. The student might have tried to imitate a model solution which has different input. The *Fix-heap-property* sequences cannot be explained systematically, but they still have the heap property holding in the end. The student has understood the end result, but not the algorithm. The *Null* submission does not have any swaps; probably the student has clicked the "Grade" button by mistake. The rest of the inexplainable submissions usually have valid father-child swaps, but their location and direction of lower value is arbitrary. The most likely explanation is that the student is just experimenting with the exercise; they might have not understood the purpose of the algorithm yet.

8 Misconception catalogue

This section contains a catalogue of misconceptions related to elementary data structures and algorithms that learners of this topic have, based on computer science education literature. The topics are grouped into the following categories: *BinTree* (binary trees and binary search trees), *Efficiency* (general time and space efficiency), *Heap* (binary heap) *LList* (linked list), *Recursion*, *RedBlack* (Red-black tree), and *Sorting*.

No.	Category	Description	Reference
1	BinTree	Every binary tree is also a search tree.	[4, p. 23]
2	BinTree	To create a balanced binary search tree, one must begin with the median value and then alternatingly insert decreasing and increasing values from the median. Example input claimed as correct: 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15.	[31, p. 172]
3	BinTree	Off-by-one error choosing 7 as the root in misconception 2.	[31, p. 172]
4	BinTree	Inserting keys in a random order into a binary search tree guarantees a balanced tree.	[31, p. 172]
5	BinTree	If a binary search tree contains N values, the root currently holds the median. If then N more values greater than root are added, the root still holds the median.column	[31, p. 172]
6	BinTree	Adding the Nth value can take time proportional to N^2.	[31, p. 172]
7	BinTree	Adding cannot take $O(N)$ time.	[31, p. 172]
8	BinTree	Adding cannot take $O(\log N)$ time.	[31, p. 172]
9	BinTree	A binary search tree is always initially balanced.	[31, p. 175]

10	BinTree	After insertion into a nonempty binary tree, the root solely decides the resulting shape of the tree.	[31, p. 175]
11	BinTree	Binary tree is always the fastest way to implement a *generalised array*, where arbitrary integer key–value pairs can be added and searched by key.	[31, p. 173]
12	Efficiency	Ignore performance when choosing one of given data structures; only consider which ones are capable of implementing the requirements.	[31, p. 173]
13	Efficiency	A shorter program, in terms of code lines, is more efficient.	[7, 32]
14	Efficiency	The fewer variables, the more time-efficient the program.	[7, 32]
15	Efficiency	Two programs containing the same statements, but in different order, are equally efficient.	[7, 32]
16	Efficiency	Two programs that perform the same task are equally efficient.	[7, 32]
17	Heap	Build-heap is only done in an array.	[25, p. 31]
18	Heap	Build-heap is only done in a binary tree.	[25, p. 31]
19	Heap	*No-recursion.* Subprocedure MIN-HEAPIFY in BUILD-MIN-HEAP (algorithm **??** does not call itself recursively.	[27]
20	Heap	*Heapify-with-Father.* Both children of node i are compared with their parent separately and swapped if necessary in subprocedure MIN-HEAPIFY of BUILD-MIN-HEAP (algorithm **??**).	[27]
21	Heap	*Delayed recursion.* Recursive heapify operations are executed only after one level of the binary-tree representation or the whole heap array has been traversed.	[27]
22	Heap	*Left-to-right.* Each level of the heap is traversed from left to right instead of right to left, as it is with decreasing index i in algorithm **??**.	[27]
23	Heap	*Smallest-instantly-up* Start from the root node and traverse top-down and left-to-right (increasing i in algorithm **??**). At each step, swap the current node and its smallest child, if necessary.	[27]
24	LList	When a new item is added to the end of a linked list, ther tail pointer does not need to be updated.	[31, p. 171]
25	LList	A linked list must be traversed from the beginning to the end regardless of the tail pointer.	[31, p. 171]
26	LList	Binary search can be efficiently used on a sorted, doubly-linked list.	[31, p. 171]
27	LList	Searching simultanously from both ends of a doubly-linked list improves performance.	[31, p. 171]

28	LList	To implement an efficient data structure for an undo buffer with a singly-linked list, LIFO property or performance is not important.	[31, p. 174]
29	Recursion	One should always have two subsets on each step in a divide-and-conquer algorithm.	[25, p. 32]
30	Recursion	*Looping model.* Recursion is a way to loop through an array of items.	[8]
31	Recursion	*Active model.* There is a recursion tree with recursive flow of control, but the solution is calculated at the base case.	[8]
32	Recursion	*Step model.* Recursion proceeds only one recursive step after a conditional expression and then it might reach the base case.	[8]
33	Recursion	*Return value model.* The return value of the next recursion instance can be used before the next recursion instance is completed.	[8]
34	Recursion	*Magic model.* Student recognises a recursive program, but does not understand how it precisely forms its solution.	[8]
35	Recursion	*Algebraic model.* Student interprets a recurrence relation as an algebraic equation.	[8]
36	Sorting	Unnecessary swaps in student-written sorting algorithms.	[29]
37	Efficiency	The upper and lower bound of asymptotic complexity can be different even if the the exact cost function is known.	[22, 6] [2]
38	Efficiency	Confusion on the use of notations for upper bounds (Big-O), tight bounds (Big-Θ), and lower bounds (Big-Ω).	[22, 6]
39	Efficiency	An upper bound is the worst case, or vice versa.	[22, 6]
40	Efficiency	A lower bound is the best case, or vice versa.	[22, 6]
41	Efficiency	Tight bound is the average case, or vice versa.	[22, 6]
42	Efficiency	The best (worst) case for an algorithm occurs when the input size is as small (large) as possible.	[22, 6]
43	Efficiency	The upper and lower bounds of an algorithm are the same as the the upper and lower bounds of a problem.	[22, 6]
44	Efficiency	Poor intuition for relative differences for common growth rates of algorithms, such as n^2 versus $n \log n$ versus n.	[6]
45	Efficiency	Weak understanding of logarithms. For example, not recognising that n versus $\log n$ has the same relationship as 2^n versus n.	[6]

[2] The page reference for 22 here is `https://github.com/OpenDSA/OpenDSA/blob/master/RST/en/AlgAnal/AnalMisunderstanding.rst`.

46	Efficiency	Confusion of addition and multiplication when analysing loops. Example: computing a cost as $n!$ when the correct answer is $\sum_i^n = O(n^2)$.	[6]
47	Efficiency	Loops always have cost that is linear on the maximum size of the control variable. (Counterexample: loop control variable is doubled or halved per iteration.)	[6]
48	Efficiency	An algorithm with fewer loops is more efficient.	[6]
49	Efficiency	Confusing the best, average, and worst case scenarios of a particular algorithm.	[6]
50	RedBlack	Consider number of black nodes only on paths from root to a visible, nonempty leaf node when coloring a red-black tree.	own work
51	Heap	*Wrong-duplicate.* If a node of a minimum binary heap is greater than its children, and both of the children are equal, swap the node with its right child.	[10]

9 Miscellaneous

References

[1] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM*, 26(9):677–679, 1983.

[2] Jeffrey Berk. The state of learning analytics. *T+D*, 58(6):0 – 39, 2004.

[3] D. Dagger, A. O'Connor, S. Lawless, E. Walsh, and V. P. Wade. Service-oriented e-learning platforms: From monolithic systems to flexible services. *IEEE Internet Computing*, 11(3):28–35, 2007.

[4] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 21–26, New York, NY, USA, 2012. ACM.

[5] Shahira El Alfy, Jorge Marx Gómez, and Anita Dani. Exploring the benefits and challenges of learning analytics in higher education institutions: a systematic literature review. *Information Discovery and Delivery*, 47(1):25–34, 2019.

[6] Mohammed F. Farghally, Kyu Han Koh, Jeremy V. Ernst, and Clifford A. Shaffer. Towards a concept inventory for algorithm analysis topics. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 207–212, New York, NY, USA, 2017. ACM.

[7] Judith Gal-Ezer and Ela Zur. The efficiency of algorithms—misconceptions. *Computers & Education*, 42(3):215 – 226, 2004.

[8] Tina Götschi, Ian Sanders, and Vashti Galpin. Mental models of recursion. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 346–350, New York, NY, USA, 2003. ACM.

[9] Ville Karavirta, Petri Ihantola, and Teemu Koskinen. Service-oriented approach to improve interoperability of e-learning systems. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 341–345. IEEE, July 2013.

[10] Ville Karavirta, Ari Korhonen, and Otto Seppälä. Misconceptions in visual algorithm simulation revisited: On ui's effect on student performance, attitudes, and misconceptions. In *2013 Learning and Teaching in Computing and Engineering*, pages 62–69. IEEE, March 2013.

[11] N.N.M. Kasim and F. Khalid. Choosing the right learning management system (lms) for the higher education institution context: A systematic review. *International Journal of Emerging Technologies in Learning*, 11(6):55–61, 2016.

[12] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] Donald E. Knuth. *The art of computer programming*, volume Volume 1 / Fundamental Algorithms. Addison Wesley Longman, Reading, Massachusetts, 2000.

[14] Ari Korhonen. *Visual Algorithm Simulation*. PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, 2003.

[15] Ari Korhonen, Otto Seppälä, and Juha Sorva. Automatic recognition of misconceptions in visual algorithm simulation exercises. In *2015 IEEE Frontiers in Education Conference (FIE)*. IEEE, October 2015.

[16] n.d. Cs-a1140 - data structures and algorithms, 10.09.2018-12.12.2018, 2018.

[17] n.d. Cs-a1141 - tietorakenteet ja algoritmit y, 11.09.2018-10.12.2018, 2018.

[18] n.d. Jsav/src at master · vkaravir/jsav · github, 2018.

[19] n.d. A+, 2019.

[20] n.d. Architecture — a+ lms, 2019.

[21] n.d. Math.random() - javascript — mdn, 2019.

[22] n.d. Opendsa/av/development at master · opendsa/opendsa · github, 2019.

[23] n.d. Tietotekniikka 2018-2020 - teknistieteellinen kandidaattiohjelma - into, 2019.

[24] n.d. Utility functions · jsav, n.d.

[25] Wolfgang Paul and Jan Vahrenhold. Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 29–34, New York, NY, USA, 2013. ACM.

[26] Otto Seppälä. Modelling student behavior in algorithm simulation exercises with code mutation. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, pages 109–114, New York, NY, USA, 2006. ACM.

[27] Otto Seppälä, Lauri Malmi, and Ari Korhonen. Observations on student misconceptions—a case study of the build – heap algorithm. *Computer Science Education*, 16(3):241 – 255, 2006.

[28] Juha Sorva. *Visual Program Simulation in Introductory Programming Education*. PhD thesis, Aalto University, School of Science, 2012.

[29] Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, pages 83–92, New York, NY, USA, 2012. ACM.

[30] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

[31] Daniel Zingaro, Cynthia Taylor, Leo Porter, Michael Clancy, Cynthia Lee, Soohyun Nam Liao, and Kevin C. Webb. Identifying student difficulties with basic data structures. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 169–177, New York, NY, USA, 2018. ACM.

[32] Nesrin Özdener. A comparison of the misconceptions about the time-efficiency of algorithms by various profiles of computer-programming students. *Computers & Education*, 51(3):1094 – 1102, 2008.