An Algorithm for Explaining Algorithms

Tomasz Müldner
Acadia University
108 PUBLICATIONS **417** CITATIONS

TR-2003-01 November, 2003

An Algorithm for Explaining Algorithms

Tomasz Müldner
Jodrey School of Computer Science, Acadia University,
Wolfville, NS, Canada B4P 2R6
Email: tomasz.muldner@acadiau.ca

Jodrey School of Computer Science, Acadia University, Wolfville, Nova Scotia, Canada

Abstract

Algorithm visualization aims to facilitate the understanding of algorithms by using graphics and animation to reify the execution of an algorithm on selected input data. However, the pedagogical potential of visualization is lessened if irrelevant objects are included in the visualization, or if essential properties of the algorithm, such as invariants, are left out. In addition, many current visualization techniques tend to present the algorithm at a single level of abstraction, which has been shown to be less effective than if the visualization occurs at various levels of generality.

In this paper, we present a novel approach for explaining algorithms that aims to overcome these downfalls. An algorithm explanation includes abstractions organized as a tree, an explanation of time complexity, and interactive questions. Each abstraction is designed to *focus* on a single operation used directly or indirectly in the algorithm. Each abstraction provides an abstract data type, ADT, which gives a high-level view of generic data structures and operations. Operations are provided in a textual form, but there is also an associated visual description used to help the student to understand basic properties of this abstraction; for example algorithm invariants. Each ADT operation is either implemented in an abstraction at the lower level, or it is a primitive operation. There is an abstract implementation of primitive operations, which can easily be mapped to a selected programming language thereby providing specific implementation of the algorithm.

The student would study algorithms by interacting with a system for algorithm explanation, called MAK (first letters of Mukhammad ibn Musa Al-Khorezmi name). MAK is an interactive system, designed to include a student model to support evaluations and adaptations. It guides the student through levels of abstraction, and asks related questions; for example what are the invariants of the algorithm. Catalog entries are categorized using algorithmic design patterns.

1. Introduction

In the ninth century, Mukhammad ibn Musa Al-Khorezmi, the chief mathematician in the academy of sciences in Baghdad, helped establish the Indian numbering system, using decimal notation and the zero. (Al-Khorezmi apparently came from the oasis of Khorazem, at the southern end of the Aral Sea, in what is now Uzbekistan.) When his work reached Europe, the Europeans misspelled the author's name and called its use "algorism" or "**algorithm**."

Today, an *algorithm* means a specific set of instructions for carrying out a procedure or solving a problem, and algorithms play an intrinsic role in Computer Science. Because of their importance, researchers have been trying to find the best way to teach and learn algorithms. One of the best known approaches has been to use visualization; especially its understanding as "the power or process of forming a mental image of vision of something not actually present to the sight", (see Petre et al. 1998a). Software visualization has grown from early flow-charts to current sophisticated graphics workstations showing 3D visualization of complex software systems. Software visualization uses graphics to show abstractions of data, and animation to convey the temporal evolution of a computer algorithm (see Stasko & Lawrence 1998).

Visualizations use various kinds of multimedia, including graphics, animation, video and voice, (the last kind is also called auralization, see Brown & Hershberger (1998)). They come in a form of local programs executed in a single machine, or distributed, web-based programs, using various kinds of server-side scripts (such as CGI or servlets) to produce visualizations, and the remote user can use applets or Java Server Pages to access these visualizations, and choose the required algorithm, select input data, and so on.

There have been many papers describing the use of animation to software explanation (in this paper, we will not review these usages, for two best known anthologies see Gloor (1992), and Gloor (1998).

A typical approach used for algorithm visualization is:

1. take the *description* of the algorithm (usually this description comes in the form of the code in a specific programming language, such as C)
2. graphically represent *data* in the code using bars, points, etc.
3. use animation to represent the *flow of control*
4. show the animated algorithm and hope that the learner will now *understand* the algorithm

Step 2 is often automatically generated from the source code, possibly with the help of specifying "interesting events" (see Brown & Sedgewick 1998).

There are various problems with the above approach. First, providing a *graphical representation* of an algorithm is just another way to show the *code* of the algorithm – instead using a textual programming language, we use a graphical language. Executing the visualization, we *simulate* the code written in a textual language, using a graphical language and the representation is typically at the *low level of abstraction* (shows low level steps).

To *explain* an algorithm, too much emphasis may be placed on meta-tools (graphics and animation) rather than the problem at hand. Indeed, some studies found that the effect of using animation is either neutral or even negative (see Stasko & Lawrence 1998). Dijkstra (1989) even feared "*permanent mental damage for most students exposed to program visualization software*". However, apparently there are some good sides of software visualization. For example, Crosby and Stelovsky (1995) found that students who interacted with an algorithm animation performed significantly better than students who listened to a lecture. The most recent overview evaluation of the educational effectiveness of algorithm visualization is given in Hundhausen et al. (2002).

In this paper, we describe an alternative and systematic procedure to explain algorithms, and this is why we talk about an algorithm for explaining algorithms. Our approach is based on results of evaluations of existing algorithm visualizations, various findings from Cognitive Psychology and Constructivism Theory, and experience from software engineering and verification regarding the use of multiple levels of abstraction and properties, such as invariants, to explain algorithms and justify their correctness. Following the experience with design patterns, see Gamma et al. (1995), we believe that algorithm explanation should be prepared by experts, who have intimate knowledge of the algorithm and the best of coding it.

Specifically, we use a hierarchical, tree-based algorithm abstraction model. In this model, a single abstraction is designed to explain one operation using the Abstract Data Type (ADT) defined for this specific abstraction, with the top abstraction explaining the algorithm. For example, the root abstraction for a selection sort algorithm is designed to explain this algorithm, using the ADT operations. In this example, one of these operations is a function smallest() that finds the smallest element in the sequence. A visual representation is used by the student to help him or her understand the basic properties of this abstraction; for example, invariants of the selection sort. The lower levels of abstraction focus on operations that are considered primitive at the higher levels. In the selection sort example, the low level abstraction provides ADT with primitive operations to represent the function smallest(). Accordingly, some operations are implemented at lower levels of abstraction, while others are left as primitives.

The student would study algorithms by interacting with a system for algorithm explanation, called MAK (first letters of Mukhammad ibn Musa Al-Khorezmi name). MAK is an interactive system, designed to include a student model to support evaluations and adaptations. It guides the student through levels of abstraction, and asks related questions; for example what are the invariants of the algorithm.

The paper is organized as follows. In Section 2, we briefly describe related work on algorithm visualization and animation. Then, in Section 3, we describe our approach, and in Section 4, we provide several examples of algorithm explanations. Finally, in the last section, we provide some conclusions and describe future research.

Parts of this report have been revised, based on the most recent work from (Müldner and Shakshuki 2004 a) and (Müldner and Shakshuki 2004 b).

2. Background: Algorithm Visualization

For the sake of completeness, we start with several definitions (mostly borrowed from Petre et al. 1998a). An algorithm animation AA, (also called algorithm visualization, AV) provides an abstraction of the algorithm data and operations, visualizes the current state of the algorithm, and animates transitions between successive states. There is an essential difference between AA and program animation, PA (or, software animation, SA) in that the former visualizes algorithms while the latter visualizes the actual program execution.
To illustrate this point, in Figure 1, we show a typical example of an algorithm animation - a screenshot from the animation of insertion sort from Duke University, produced by Whitley & King (2001).

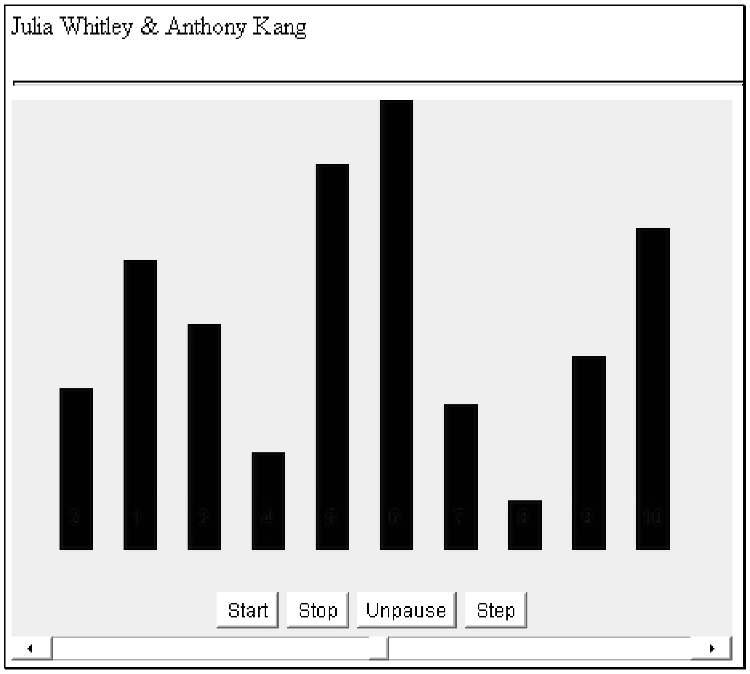


Figure 1: Animation of insertion sort

The animation is accompanied by the following instructions:
- Start by comparing the first two rectangles in the group, and shift the second rectangle to the left if it is smaller.
- Look at the next rectangle, compare with the one on its left, and shift it to the left until it is smaller than the one to its right and larger than the one to its left.
- Repeat step 2 until finished

As you can see, the user has to map the problem domain (values to be sorted) to the graphical domain (bars), and then looking at the animation the user has to retrieve essential properties of the algorithm (such as maintaining a sorted prefix). Therefore, this, and many other existing algorithm animation systems resemble visual debuggers in that they show the *execution* of the algorithm by code-stepping, work at the lowest level of abstraction, and illustrate only the primitive code. This approach constrains users to view the code in the order of execution, which is the wrong information for understanding the

algorithm and has a poor cognitive fit with the plan-and-goal structures that users are trying to extract from the code (see Petre et al. 1998a). In any case, runtime interpretation requires specific input data (cannot consider all possible inputs) and often suffers from the lack of focus on relevant data (see Braune & Wilhelm 2000).
In addition, most existing systems do not attempt to visualize or even suggest essential properties, such as invariants, which are essential for understanding algorithm correctness. One notable exception is the approach in Wilhelm et al. (2001), which uses a static source code analysis to abstractly execute the algorithm on “all possible sets of input data”, and visualize invariants. Also, there have been very few attempts to visualize recursive algorithms; the most notable example is given by Stern & Naish (2002a and 2002b).

3. Algorithm Explanation

In this section, we describe our proposal, called Algorithm Explanation, AE. The main goal of AE is to support the task of algorithm comprehension (we are interested in learning, rather than teaching). According to Petre et al. (1998b) to make this task possible, the learner has to build a mapping from the domain consisting of the algorithm entities and temporal events to learner’s conceptions of these entities and events.
AE uses a variety of tools to help the students to learn algorithms, and visualization is just one possible tools. Indeed, we provide both, textual and visual representation, and use the latter to help students recognize and understand algorithm properties. Note that Petre et al. (1998b) state that two representations are not necessarily better than one. As we will explain in Section 4, we use a visual representation to help student derive essential properties, such as invariants. However, we leave the option of using just the text, and if the students can successfully derive invariants form the text then they do not have to see the visual representation. Thus, the visual representation is used to *help* reason about the textual representation. It is available to the user, because in some cases this representation provides the so-called “gestalt” effect; it provides an overview making a structure clearer (see Petre et al. 1998b).
The target audience is students who know well programming in at least one programming language such as C, C++ or Java and are to learn algorithms. These students will have to learn our conventions used for writing the pseudocode and showing visualizations.

The next section describes what exactly we want to achieve by using AE, i.e. what do we expect the student will learn.

3.1 Goals

We believe that the following goals should be achieved:

G1. Understanding of both, *what* the algorithm is doing and *how* it works
G2. Ability to justify the algorithm correctness (*why* the algorithm works)
G3. Ability to *program* the algorithm in any programming language
G4. Understanding of *time complexity* of the algorithm

In order to describe a system that satisfies these goals, we now list requirements that an algorithm explanation should satisfy.

3.2 Requirements

To achieve the above goals, the following requirements must be satisfied:

R1. The algorithm is presented at several levels of abstraction
R2. Each level of abstraction is represented by the pseudocode
R3. Active learning supports
R4. The design helps to understand time complexity
R5. The presentation uses multiple views
R6. Presentations are designed by experts

Now, we elaborate on each the above requirements and explain how they are related to our goals.

Re R1. There are several advantages of using multiple levels of abstraction. First, the research in cognitive psychology on knowledge organization supports using multiple levels of abstraction when dealing with complex tasks; (see Anderson 1980). The idea of using more than one level of abstraction is also supported by Petre et al. (1998b) who claim that in general, it is hard to determine a *single* suitable level of abstraction. Second, the research has shown (see Petre et al. 1998b) that if the presentation is designed to highlight some kind of information, then it is likely to obscure other kinds. In our approach, each level of abstraction is used to highlight a single kind of information; for example invariants, and so the student can *focus* on this kind of information. Third, to reason about a process in the world requires setting up a mental model of each state of an algorithm (see Petre et al. 1998b).
There are two possible approaches that algorithm abstraction levels may be defined. For both approaches, the top level is first defined, using operations considered to be primitive at this level. Then, with the first approach, each top level primitive is defined at the single lower abstraction level, possibly using other primitives (those primitives would be defined at the next lower abstraction level). With the other approach, the top level primitives are replaced (rather than defined) by their definitions, which again can use some lower-level primitives. Thus, the second approach resembles code inlining. We have chosen to take the former approach, which helps to concentrate at issues pertinent to one abstraction level. We disagree with the criticism from Faltin (2001) that the result of structuring algorithms using this approach produces code that is longer and less efficient. The recent advances in compiler technology and the right mapping of the correctly designed pseudocode to a programming language make the code sufficiently efficient (see our examples in Section 4). The latter approach was investigated in Fhe (2003).
Note that in our approach, each level of abstraction has a limited number of states, which is important because the large number of states makes it difficult to reason about properties of the algorithm. We advocate top-down approach, starting from the top level of abstraction and then moving to lower levels. The reason for using a top-down

approach is that it helps students to understand various special cases, such as various ways to compare integer values in a sorting algorithm, whereas a bottom-up approach hardcodes specific cases. This approach differs from the one used in “Algorithms in Action”, AIA, from Stern & Naish (2002a and 2002b), where students were allowed to choose the starting level of abstraction, and often chose a bottom-up approach. Therefore, this requirement contributes to G1, in both, “what” and “how”.

Re R2. The pseudocode is a model of the algorithm, and it includes the high-level abstract data structures and operations. These operations are designed so that they can be directly mapped to most procedural and object-oriented programming languages. Using pseudocode, the algorithm can be studied independently of any programming language; see (Fleischer & Kucera 2001). The pseudocode given at each level of abstraction has an optional visual representation, which exposes its properties, in particular its invariants. This approach is supported by findings provided by Petre et al. (1998b) that proper explanation supports a display-based reasoning; that is the display becomes a focus for reasoning and supports creating the mental image of things that do not really appear there. For example, the pseudocode in Section 4.1.2.1, given at the highest level of abstraction, helps to recognize two invariants of this algorithm. Note that our concept of a pseudocode differs from that used in Stern & Naish (2002a and 2002b), where the pseudocode is based on C, with some abstract procedure calls. By exposing the algorithm’s properties, in particular its invariants, at various levels of abstraction, this principle contributes to G1, G2. In addition, the pseudocode is written in generic terms so that it can be used not only to write concrete implementations in specific programming languages, such as Java, but also to produce different concrete implementations (for example, using linear or binary search to implement insertion sort), which contributes to G3.

Re R3. Active learning follows Cognitive Constructivism principles (for example, see Hundhausen 2002) and this style of learning includes various kinds of interactions with the student. For example, students are able to use their own input data sets; use a do-it-yourself mode, that is predict the next of the algorithm (see Faltin 2001 or Stern 2002a and 2002b), and determine the essential algorithm properties. AE always comes with several sets of representative sample input data; the learners can also use their own input data. AE includes student evaluation, which consists of programming the target algorithm, a textual programming language. This ability is missing from all algorithm animation systems, but in our opinion it is absolutely essential. The ultimate goal of teaching algorithms is to educate programmers, who will be able to implement various algorithms. Finally, AE includes post-test (see Stasko & Lawrence 1998), with tasks such as hand-execution of the algorithm on sample sets of input data, and answering various questions about the algorithm.

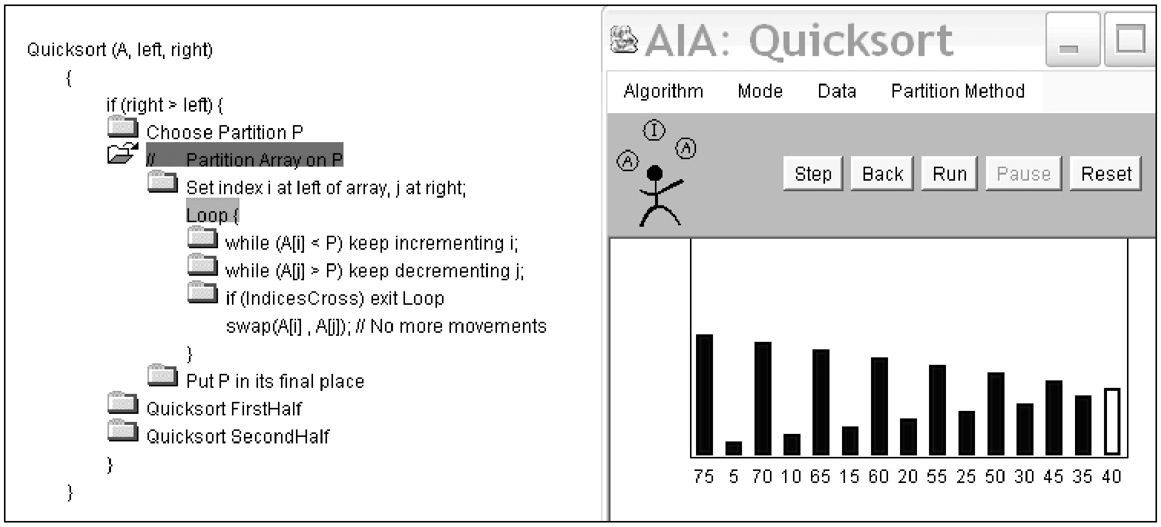
Re R4. AE helps the learner to understand time complexity by providing a program, which the user can do to input sample data and plot the function representing the time spent by the algorithm when these data are used (this approach is based on Brown & Sedgewick 1998 and Horstmann 2001). Other tools that help to understand time complexity are described in Section 3.6.

Re R5. Multiple views showing algorithm states are used to avoid forcing the viewer to remember the previous states (two consecutive frames are shown in the “comic strip" approach; see Biermann & Cole 1999). The decreased cognitive overhead contributes to understanding how the algorithm works, which contributes to goal G1.

Re R6. AE presentations are designed by experts who have a complete understanding of essential properties to be exposed (such as invariants). Essentially, our approach is similar to that used in design patterns, which are created by experts and used by novices. It is also similar to the concept of “algorithmic design patterns”, described in Goodrich & Tamassia (2001). The design by experts leads to the better code, see Section 4.1.3. Note that our proposal differs from that proposed by some (see Hundhausen et al. 2002) who recommend that students should design algorithm animations. We believe that designing algorithm animations concentrates on meta-tools and distracts students from their primary goal, which is learning algorithms.

3.3. Related Work

Since there are some similarities between our approach and AIA developed by Stern & Naish (2002a and 2002b), let us now compare these two approaches. Figure 2, based on AIA, shows two windows during the execution of quick sort (there two other windows, not shown here, to display context-sensitive help and a textual description of the algorithm). The pseudocode in the left window contains parts that can be unfolded to show a lower level of abstraction. Since it well known that users tend to click any “clickable” objects on the screen, it is likely that this is why evaluations of this system found that students often start from the low level. Note that AIA is not using an abstract, language-independent pseudocode, which in our approach can be mapped to various programming languages. It is our conclusion that AIA does not help to achieve the goal G2, and has some features that belong to a visual debugger rather than to a tool to learn algorithms.



```
Quicksort (A, left, right)
    {
        if (right > left) {
            Choose Partition P
            //      Partition Array on P
                Set index i at left of array, j at right;
                Loop {
                    while (A[i] < P) keep incrementing i;
                    while (A[j] > P) keep decrementing j;
                    if (IndicesCross) exit Loop
                    swap(A[i] , A[j]); // No more movements
                }
                Put P in its final place
            Quicksort FirstHalf
            Quicksort SecondHalf
        }
    }
```

Figure 2: Quicksort from AIA

3.5 AE Descriptions

Our goal is to develop an AE catalog, consisting of AE descriptions of many well known algorithms. Each description, or *catalog entry*, consists of the following four parts:

1. A hierarchical Abstract Algorithm Model (AAM) consists of abstractions representing operations. Each abstraction explains a single operation op(), and consists of a textual representation and a visual representation. The textual representation includes an ADT that provides data types and operations. It also provides a representation of the operation op() using the ADT from this abstraction. All these abstractions form a tree, rooted at the abstraction representing the algorithm operation.
2. An intermediate representation of the so-called primitive operations from the ADTs (which are not implemented in the AAM). This representation is designed to help the student to write concrete implementations.
3. Tools that can be used to help predicting the algorithm complexity.
4. Questions for students, including "do it yourself" mode.

To explain the first part, let's assume that f() is an operation. The abstraction that explains f(), abst(f) is a pair (ADT, representation of f() in the ADT), where ADT consists of data types and primitive operations. An abstraction abst(f) is a parent of an abstraction abst(g) if g is one of the primitive operations from the ADT abst(f). Therefore, a child abstraction provides a partial implementation of the operation from the parent abstraction. Typically, there are only few operations from any abstraction's ADT that are implemented in a child of this abstraction; others are considered primitive operations. An Abstract Algorithm Model (AAM) of an algorithm f() is a tree rooted at abst(f). For example, the AAM of a selection sort is a tree of abstractions rooted at abst(selection).

To explain an algorithm, we construct an AAM tree with sufficient number of levels so that the student is able to understand how and why the algorithm works. In particular, the student can form and justify invariants of the algorithm.

The second part provides the intermediate representation of all AAM's primitive operations. To implement the algorithm in a specific programming language, the student has to map to the selected language all primitive operations that do not have implementations in the AAM. This representation is called an Abstract Implementation Model (AIM). The representations in AIM are generic in that they are not using any specific programming language; instead they use high-level concepts that can be mapped to many languages. Once all primitive operations are represented in the AIM, the code for the algorithm and possibly some of its operations can be easily represented using this AIM.

3.6 Design of MAK

MAK is an interactive system, which allows the student to select one of the available algorithms to study, and then it uses a student model to record student activities. (Currently, MAK is only designed and its implementation is to begin this winter.) These interactions are vital to provide support the active learning model.

MAK helps the student to select one algorithm, and it guides this student through the AAM by following these steps:

1) The root abstraction is explained:
   a) The ADT associated with the root abstraction is shown
   b) the implementation of the operation from this abstraction, using the ADT operations is shown
   c) the students is asked to explain basic properties of the implementation from b), including invariants
   d) the student may choose to see the visualization associated with the implementation from b); if so this visualization is made available, and the student can enter specific input data and watch the visualization, or use a "do it yourself " mode to test their understanding of the algorithm
   e) MAK explains which ADT operations are primitive.

2) For any child abstraction, associated with the higher-level operations that are not primitive, the explanation process from 1) above is repeated.

3) Now, the student is supposed to implement an algorithm in a selected programming language. First, MAK shows AIM, and asks the student to implement an algorithm in one of several available programming languages (which are available internally to MAK, but not shown to the student). MAK also provides sample input data, including boundary cases that can be used by the student for testing and debugging their implementations.

4) Explanation of algorithm complexity is one of the most difficult goals of algorithm visualization, because it requires mathematical proofs that are hard to visualize. The current version of MAK includes three kinds of tools designed to help the student to derive the complexity of the algorithm being studied. The first tool, based on Horstmann (2001), gives the student a chance to experiment with various data sizes and plot a function that approximates the time spent on execution with these data. The second tool, based on Goodrich & Tamassia (2001) p. 477, provides a visualization that helps to carry out time analysis of the algorithm. Finally, the third tool asks students various questions regarding the time complexity, and questions specific to the algorithm being studied, and evaluates their answers.

In general, AV systems can be categorized using a taxonomy introduced by Price et al. (1998). Using this taxonomy, MAK is a *specific* system (used only for limited number of algorithms), and its *content*, which defines what aspect of an algorithm is visualized, concentrates on explaining algorithm properties. The *form*, which is how a system is presented, is text, graphics, animation and possible voice; and the *method*, which describes how to develop an explanation, is currently hand-coding from scratch. Finally, the *interactions* include describing invariants, answering specific questions, and implementing the algorithm.

Various kinds of algorithms require different approach, in particular different visualizations. Iterative algorithms can be explained using an operational semantics (state, transition, next state), from which the student is supposed to derive the code and invariants. Recursive algorithms require a different treatment. Therefore, to assess our approach, in the following section we consider AE catalog entries for three sorting algorithms; two iterative algorithms, insertion sort, and selection sort, and one recursive algorithm, quick sort.

4. Examples of Algorithm Explanation Descriptions

The following three sections show partial descriptions of the catalog entries for the three sorting algorithms. Note that we show visualizations of execution of these algorithms for specific input data, but MAK allows the student to enter other input data, and show the visualization with these data.

4.1 Selection sort

In this section, we describe the selection sort, see Aho et al (1983). The goal of the selection sort algorithm is to sort the sequence according to the given comparator, therefore we consider:

```
void selection(Seq<T> t, Comparator comp)
```

An algorithm is explained using various levels of abstraction including root and leaf levels of abstractions. Recall that each abstraction is designed to present a single operation used in the algorithm. In this example, AAM consists of two abstractions.

4.1.1 Root Abstraction

The root abstraction provides the ADT and the representation of the algorithm using this ADT. This abstraction is concerned with the selection sort, i.e. it can be described as `abst(selection)`.

Abstract Data Type

The ADT consists of data and operations. The data consists of sequences of elements of type T, denoted by Seq<T>, with a linear order. In general, this order can be defined in one of three ways:

1. Type T supports the function int compare(const T x)
   so that y.compare(x) returns -1 if x is less than y, 0 if they are equal and +1 otherwise.
2. Type T supports the "<" operation, which can be used to compare two elements of this type.
3. There is a function (or a type)
   int comparator(const T x, const T y)
   so that comparator(x, y) returns -1 if x is less than y, 0 if they are equal and +1 otherwise.

Note that the above model is generic, in particular it does not specify a concrete ordering relation, such as "<" used for real numbers. Also, it does not specify any concrete data structures (for example, we can use arrays or linked lists) nor it describes details of the algorithm (for example, whether the algorithm works "in place" or with copying).

The following six operations on an element t of the domain Seq<T> are available:

- prefix(t), representing (a possibly empty) prefix of the sequence t
- inc(prefix(t)), which increments a prefix by one element;
- suffix(t), where a prefix followed by the suffix is equal to the entire sequence t;
- first(suffix), which returns the first element of the suffix;
- T smallest(seq<T> t, Comparator comp), which finds the smallest element in the sequence t (using comp);
- swap(T el1, T el2), which swaps el1 and el2.

Algorithm Representation

The goal of the selection sort algorithm is to sort the sequence according to a given comparator. The implementation of the selection sort using the ADT described above is shown in Figure 3.

```
void selection(Seq<T> t, Comparator comp) {
  for(prefix(t) = NULL; prefix(t) != t; inc(prefix(t)))
    swap( smallest(suffix(t), comp), first(suffix(t)));
}
```

Figure 3. The representation of the selection sort.

The student is provided with the code shown in Figure 3, which represents the selection sort implemented using ADT introduced in the root abstraction. Then, she or he is asked to determine the invariants of this code. There are two invariants that can be extracted from this example, including:

1) All elements in the prefix are smaller (according to the "comp" relation) than all elements in the suffix.
2) The prefix is sorted.

The first invariant is true because in each step of the “for” statement, the smallest element in the suffix is placed at the beginning of the suffix, and then the prefix is incremented, effectively placing this element at the end of the prefix. Once this invariant is established, the second invariant easily follows: appending the element from the suffix to the end of the prefix maintains the “sortedeness” of the prefix. Since the algorithm stops when the prefix contains all the elements in the sequence, the second invariant shows the correctness of this algorithm – it indeed sorts the sequence. It may be hard for the student to find both of these invariants and if they wish so, they can use the associated visualization. Figure 4 shows conventions used in the visualization.

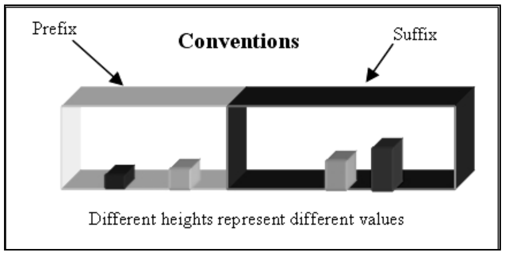


Figure 4. Conventions used by the visualization

Figure 5 shows all the states resulting from the execution of the code from Figure 3; using a five sample input data. The small arrow shows the smallest element in the suffix. States without this arrow represent “in-between” states, after swap has been completed, but before the prefix has been incremented.

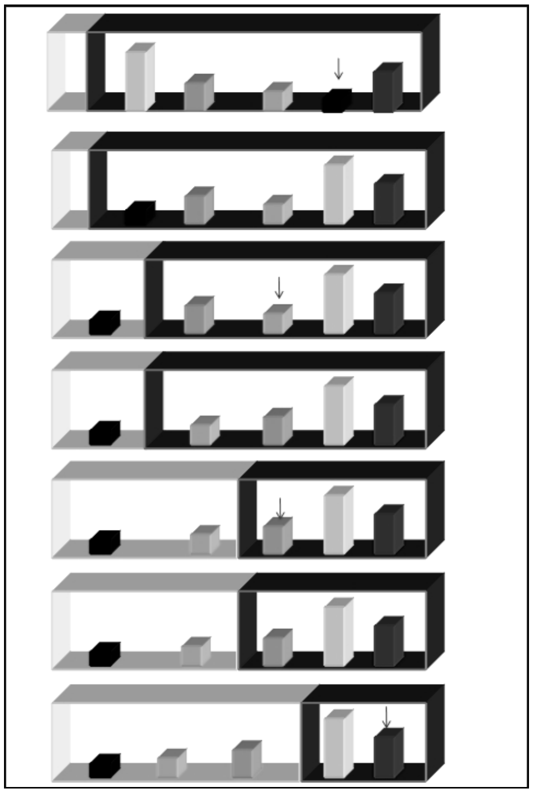


Figure 5. Partial visualization of selection sort at the root abstraction

4.1.2 Leaf Abstraction

Leaf abstractions represent various operations from the ADT described above. For this ADT, there may be as many as six abstractions that represent the children of the root abstraction; each abstraction focuses on a single operation from the ADT. Most of these operations however are intuitively obvious. Therefore, this AAM will have only one child of the root abstraction associated with the operation smallest(). Other ADT operations are considered primitive. This abstraction explains the function smallest().

Abstract Data Type

The ADT consists of data described above, and the following operations:

- first(t), which returns the first element of the sequence t;
- next(current, t), which returns the element of the sequence t, following current, or NULL if there is no such element.

Algorithm Representation

Figure 6 shows the implementation of the function smallest() using the ADT described above.

```
T smallest(Seq<T> t, Comparator comp) {
  smallest = current = first(t);
  while(next(current, t) != NULL)
     if(comp(smallest, current) < 0)
       current = smallest;
}
```

Figure 6. The representation of the function smallest.

Visualization

In Figure 7, we use a small arrow to point to the currently found smallest element, and another small arrow to point to the current element.

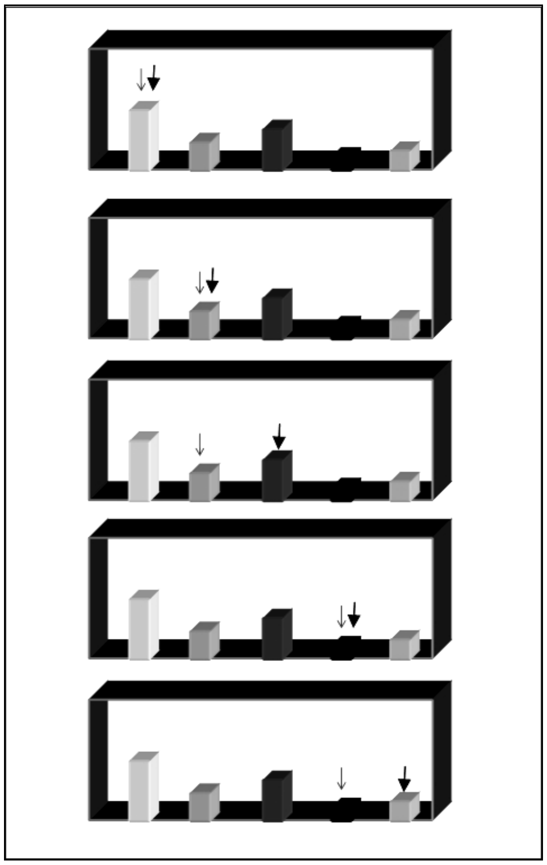


Figure 7. Visualization of smallest()

4.1.3 Abstract Implementation

This section describes a specific AIM, called the Abstract Iterator Implementation Model (AIIM). Recall that the goal of this model is to provide the intermediate representation of all primitive operations from AAM. For the purpose of our example that uses the selection sort, this model has to provide representations for five ADT operations from the root abstraction and two operations from the child abstraction. AIIM assumes that there is an Iterator type. This Iterator helps to avoid exposing the underlying representation of the object to the client and provides a sequential *access* to a data object it points to. Iteration is performed over a range, which is a half-closed interval `[a, b)`. Here, b can only be used for comparison purposes and may not be accessed. Iterator type supports the following operations:

- two iterators can be compared for equality and inequality;
- an iterator can be moved forward by executing the operation inc();
- an iterator can be dereferenced to access the value it points to.

In addition, the AIIM makes several assumptions about the ADT described above; specifically:

- the following type is available: Seq<T>::Iterator
- if t is a sequence in Seq<T>, then the following three operations are defined:

Iterator t.begin()	points to the beginning of t
Iterator t.end()	points beyond t

swap(i1, i2) swap two iterators

Table 1 shows how the seven primitive operations are represented in this model. In this table, eop and current stand for variables of type Seq<T>::Iterator.

Table 1. The representation of the primitive operations

Abstract Operation	**Representation**
prefix(t)	[t.begin(), eop)
inc(prefix(t))	inc(eop)
suffix(t)	[eop, t.end())
first(suffix)	Value of eop
swap(T el1, T el2)	Swap values of iterators respectively pointing to el1 and el2
first(t)	Value of t.begin()
next(current, t)	inc(current)

Figures 8 and 9 show the AIIM implementation of respectively the function smallest() and the selection sorting algorithm.

```
T smallest(Seq<T> t,   Comparator comp) {
  Iterator small = value of t.begin();
  Iterator current;
  for(current = t.begin(); current != t.end(); inc(current))
    if(comp(value of current, value of small)< 0)
         small = current;
   return value of small;
}
```

Figure 8. Code for the function smallest using AIIM.

```
void selection(Seq<T> t,  Comparator comp) {
   Seq<T>::Iterator eop;
      for(eop = t.begin(); eop != t.end(); inc(eop))
         swap(smallest(eop, t.end(), comp), eop);
}
```

Figure 9. Code for the selection sort using AIIM**.**

The following three sections show how the code that implements the operations used in the selection sort can be mapped to three different programming languages; specifically C, C++ and Java.

Implementation in C

An iterator in C language can be represented by a pointer. For the code presented in Figures 8 and 9, a complete C implementation is shown in Figure 10, and the main program using this implementation is shown in Figure 11.

```
static T* smallest(T* const first, T*
       const last,  int comp(const T, const T)) {
   T *small = first;
   T *current;
   for(current = first; current != last; ++ current)
      if(comp(*current, * small) < 0) s = current;
   return small;
}
static void swap(T* const a, T* const b) {
  T x = *a;
  *a = *b;
  *b = x;
}
void selection(T *x, T* const end,
    int comp(const T, const T)) {
  T* eop;
  for(eop = x; eop != end; ++eop)
    swap( smallest(eop, end), eop );
}
```

Figure 10. Complete C implementation of selection sort.

```
/* A part of the main function */

typedef struct {
  int a;
  int b;
} T;
#define SIZE(x) (sizeof(x)/sizeof(T))
T x[ ] = { {1, 2}, {3, 7}, {2, 4}, {11, 22} };
#define S SIZE(x)
int comparator1(const T x, const T y) {
  if(x.a == y.a) return 0;
  if(x.a < y.a) return -1;
  return 1;
}
int comparator2(const T x, const T y) {
  if(x.a == y.a) return 0;
  if(x.a > y.a) return -1;
  return 1;
}
```

```
void show(T *x, int size) {
 T *p;
 for(p = x; p != x + size; ++p)
   printf("(%d %d)\n", p->a, p->b);
}

int main() {
  printf("original sequence\n");
  show(x, S);
  selection(x, x+S, comparator1);
  printf("sequence after first sort\n");
  show(x, S);
  selection(x, x+S, comparator2);
  printf("sequence after second sort\n");
  show(x, S);
}
```

Figure 11. Main Program for the Complete C implementation of selection sort.

Implementation in C++

Iterators in C++ can be easily represented using STL. Figure 12 shows a complete C++ implementation of the code presented in Figure 8. The function smallest has an immediate implementation using the STL function min_element.

```
template <typename Iterator, typename Predicate>
  void selection(Iterator first, Iterator last, Predicate comp) {
    Iterator eop;  // end of prefix
    for(eop = first; eop != last; ++eop)
       swap(*min_element(eop, last, comp), *eop);
  }
```

Figure 12. Complete C++ implementation of selection sort.

Implementation in Java

An iterator in Java can be represented using the Iterator class from Java Collection classes. Figure 13 shows a complete Java implementation of the code presented in Figures 8 and 9.

```
public static void selection(List aList, Comparator aComp) {
  for (int i = 0; i < aList.size(); i++)
    swap(smallest(i, aList, aComp) , i, aList);
}
private static int smallest(int from, List aList,
        Comparator aComp) {
  int minPos = from;
  int count = from;
```

```
 for (ListIterator i = aList.listIterator(from);
               i.hasNext(); ++count)
   if (aComp.compare(i.next(), aList.get(minPos)) < 0)
     minPos = count;
  return minPos;
}
 private static void swap(int i, int j, List aList) {
  Object itemp = aList.get(i);
  Object jtemp = aList.get(j);
  aList.set(i, jtemp);
  aList.set(j, itemp);
}
```

Figure 13. Complete Java implementation of selection sort.

4.1.4 Post Test

In this section, we provide a list of questions specific for the selection sort algorithm.

Questions
1. What is the number of comparisons and swaps performed when selection sort is executed for:
a) sorted sequence
b) sequence sorted in reverse
2. What is the time complexity of the function isSorted(t), which checks if t is a sorted sequence?
3. Hand-execute the algorithm for a sample set of input data of size 4.
4. Hand-execute the next step of the algorithm for various states.
5. What’s the last step of the algorithm?
6. There are two invariants of this algorithm; explain which one is essential for the correctness of swap(smallest(), eop), and explain why.
7. Use “do it yourself“ mode.

4.2 Insertion sort

In this section, we describe the insertion sort, see Aho et al (1983). We do not include the explanation of time complexity and a post test, since these parts of this description are almost identical to that given in the previous section.
The goal of the insertion-sort algorithm is to sort the sequence according to the given comparator, therefore we consider:

void insertion(Seq<T> t, Comparator comp)

4.2.1. Root Abstraction

The ADT has the same data as the ADT in Section 4.1, and the following operations on the element t of the domain Seq<T>:
The following three operations on an element t of the domain Seq<T> are available:

- prefix(t), representing (a possibly empty) prefix of the sequence t
- inc(prefix(t)), which increments a prefix by one element;
- insert(seq<T> prefix, Comparator comp)
  inserts the element el, which immediately follows the prefix, to the sorted prefix so that it continues to be sorted.

Algorithm Representation

The goal of the selection sort algorithm is to sort the sequence according to a given comparator. The implementation of the selection sort using ADT described above is shown in Figure 14.

```
void insertion(Seq<T> t, Comparator comp) {
  for(prefix = NULL; prefix != t; inc(prefix))
    insert(prefix, comp);
}
```

Figure 14. The representation of the selection sort.

Here, the student will be used to describe an invariant of this code. The student can get help, using the following visualization.

Visualization

In Figure 15, we show all states resulting from the execution of the above code, using a sample set of five input data.

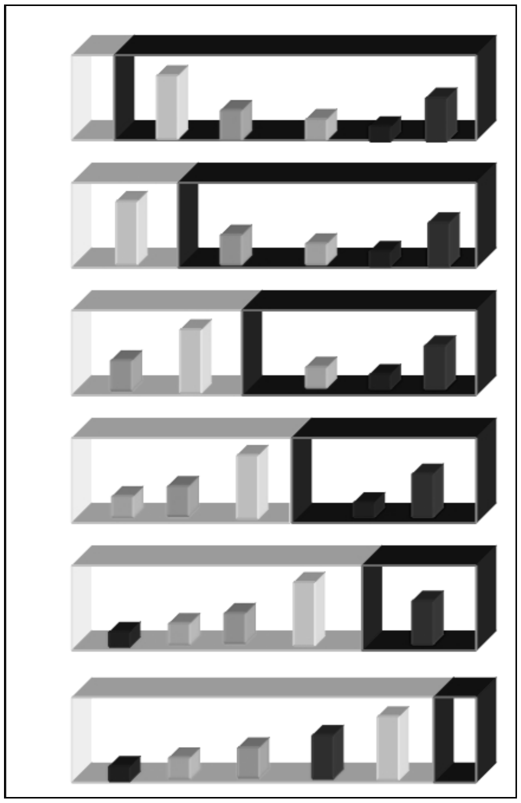


Figure 15. Visualization of Insertion sort

4.2.2 Leaf Abstraction

This AAM will have only one child of the root abstraction associated with the operation insert(). Other ADT operations are considered primitive. Therefore, the leaf abstraction explains the function insert().

Abstract Data Type

The ADT for the leaf abstraction has the same data as the ADT from Section 4.1.1, but it has the following operations:
The ADT consists of data described above, and the following operations:

- last(t), which returns the last element of the sequence t;
- prev(current, t), which returns the element of the sequence t, preceding current, or NULL if there is no such element.
- swap(T el1, T el2), which swaps el1 and el2.

Algorithm Representation

Figure 16 shows the implementation of insert() using the ADT described above.

```
void insert(Seq<T> t, Comparator comp) {
  for(current = last(t);  prev(current) != NULL; current = prev(current, t))
    if(comp(current, prev(current)) < 0
      swap(current, prev(current));
   else return;
}
```

Figure 16. The representation of the function insert.

Visualization

In Figure 17, we use two small black arrows to point to elements, which are out of order and will be swapped.

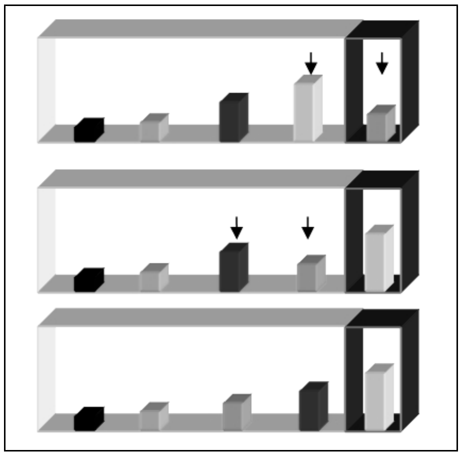


Figure 17. Visualization of insert()

4.2.3 Abstract Implementation

Here, we use the abstract implementation from Section 4.1.3, but we additionally assume that an iterator can be moved backward by executing the operation prev(). Table 2 shows how the five primitive operations (two from the root abstraction and three from the leaf abstraction) are represented in this model. In this table, eop and current stand for variables of type Seq<T>::Iterator.

Table 2. The representation of the primitive operations

Abstract Operation	Representation
prefix(t)	[t.begin(), eop)
inc(prefix(t))	inc(eop)
swap(T el1, T el2)	Swap values of iterators respectively pointing to el1 and el2
last(t)	Value of prev(t.end())
prev(current, t)	prev(current)

Figures 8 and 9 show the AIIM implementation

The code implemented using this representation is shown in Figure 18.

```
void insertion(Seq<T> t, Comparator comp) {
  Iterator eop;
  for(eop = NULL; eop != t; ++eop)
    insert(t.begin(), eop, comp);
}

void insert(Seq<T> t, Comparator comp) {
  Seq<T>::Iterator current;
  for(current = t.end(); current != t.begin(); current=prev(current))
    if(comp(value of current, value of prev(current)) < 0)
      swap(value of current, value of prev(current));
    else break;
}
```

Figure 18. Code for the functions insert and insertion using AIIM.

Below, we show how this code can be mapped to C and C++ to implement insertion sort.

4.2.3.1 C Implementation

```
void insert(T* const begin, T* const end, int comparator(const T, const T) ) {
  T* i;
  for(i = end; i != begin; --i)
    if(comparator(*i, *(i-1)) < 0)
      swap(i, i-1);
    else break;
}

void insertion(T*x, T* const end,  int comparator(const T, const T)) {
  T* eop;
  for(eop = x; eop != end; ++eop)
    insert(x, eop, comparator);
}
```

Figure 19. Complete C implementation of insertion sort.

4.2.3.2 C++ Implementation

```
template <typename Iterator, typename Predicate>
void insert(Iterator first, Iterator last, Predicate compare) {
  Iterator i;
  for(i = last; i != first; --i)
    if(compare(*i, *(i-1)))
      swap(*i, *(i-1));
    else break;
}

template <typename Iterator, typename Predicate>
// Iterator is a model of Random Access Iterator
void insertion(Iterator first, Iterator last, Predicate compare) {
  Iterator eop;
  for(eop = first; eop != last; ++eop)
    insert(first, eop, compare);
}
```

Figure 20. Complete C++ implementation of insertion sort.

4.3 Quick Sort

QuickSort, see Aho et. al (1983), is a recursive algorithm, based on a divide-and-conquer algorithmic design pattern. In this section, we will only show the code and the visualization of this algorithm.
The ADT has the same data as the ADT described in Section 4.1, and the following operations on the element t of the domain Seq<T> are available:

- T choosePivot(seq<T> t)
  selects a single element of t
- divide(T pivot, seq<T> t, seq<T> t1, seq<T> t2, seq<T> t3, Comparator comp)
  stores in t1 all elements of t which are less (according to comp) than pivot, in t2 all elements of that are equal to pivot, and in t3 all remaining elements
- void concatenate(seq<T> t, seq<T> t1, seq<T> t2, seq<T> t3)
  stores in t all elements of t1, followed by elements of t2, followed by elements of t3.

The code for the quicksort using this operations is shown in Figure 21.

```
 void quick(Seq<T> t, Comparator comp) {
   if( size(t) <= 1) return;
   pivot = choosePivot(t);
   divide(pivot, t, t1, t2, t3, comp);
   quick(t1, comp);
   quick(t3, comp);
   concatenate(t, t1, t2, t3);
}
```

Figure 21. Quick sort

Visualization

Visualization of recursive algorithms is much harder than visualization of iterative algorithms. A standard approach it to show a call tree, with a root representing the first call, and leaves representing stop conditions in the recursion (for example see the quick sort visualization in Goodrich and Tamassia (2001), pp. 469-471). Another standard approach is to show the runtime stack resulting from the recursive calls, see Stern and Naish (2002a and 2002b). We believe that neither of these two approaches is helpful, because to does not focus on recursive properties. To understand, and possibly to prove recursive definitions, one uses a mathematical induction, which involves the first step, and the recursive step. Therefore, we designed a new visualization of recursive calls, using a modified UML transition diagrams, see Fowler (2000).
Our conventions are explained using Figure 22, which represents a recursive routine to print the list backwards. A history of each call is represented by one or more boxes, placed vertically and connected by a dotted line. A name of the procedure being called labels the arrow going from left to right and pointing to the top of the first box in the history of this call. An arrow going from the right to the left and pointing to the top of the

last box represents a return from this procedure. Dotted arrows and shaded boxes represent recursive steps that are now shown by the visualization (therefore, we show only the recursive step, rather than the entire call tree). A bold arrow on the left hand side represents time; for example print(L) occurred before print(L.next()).

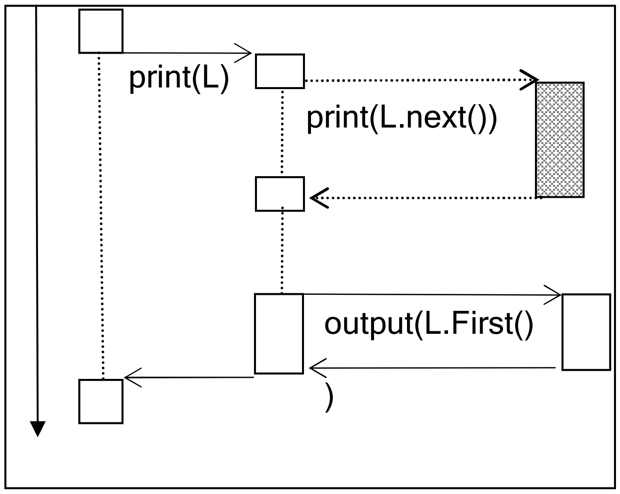


Figure 22. Conventions used for visualization of recursion.

Sequence Diagrams Visualization of Call Graphs

In Figure 23, we show a visualization of quicksort using sample input data (because of space limitations, we do not show choosePivot()). The visualization consists of two parts: the stop condition, and the recursive step.

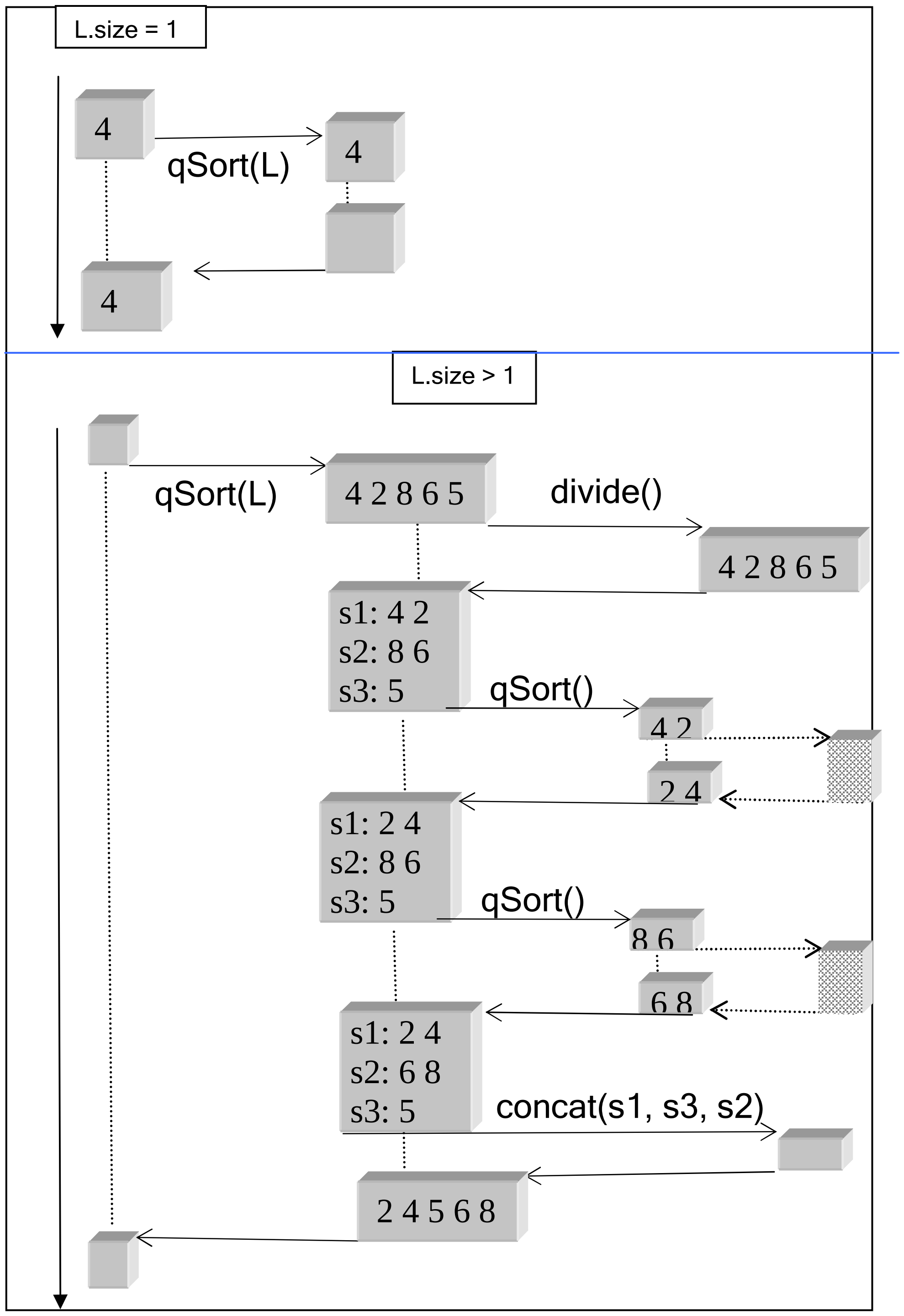


Figure 23. Sequence diagram for quicksort()

5. Conclusions and Future Research

This paper described work in progress, and there are many issues to be researched; including:

- evaluation

- student model
- different visualizations
- more complex algorithms
- generic approach

Acknowledgments
Parts of this report have been revised, based on the most recent work from (Müldner and Shakshuki 2004 a) and (Müldner and Shakshuki 2004 b).

References

Aho, A. V., Hopcroft, J.E., & Ullman, J. D (1983). *Data Structures and Algorithms*. Addison-Wesley, Reading, Mass.
Anderson, J. R. (1980). *Cognitive psychology and its implications.* San Francisco: Freeman.
Biermann, H. and Cole, R., (1999). Comic Strips for Algorithm Visualization, *NYU Technical Report 1999-778*.
Braune, B., & Wilhelm, R. (2000). Focusing in Algorithm Explanation. *IEEE Transactions on Visualization and Computer Graphics* 6(1), (pp. 1-7).
Brown, M.H., & Hershberger, J. (1998). Program Auralization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 137-144). MIT Press.
Brown, M.H., & Sedgewick, R., (1998). Interesting Events. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 155-172). MIT Press.
Crosby, M. E., & Stelovsky, J. (1995). From multimedia instruction to multimedia evaluation. *Journal of Educational Multimedia and Hypermedia* 4, (pp. 147-162).
Dijkstra, E.W. (1989). *On the cruelty of really teaching computing science*. The SIGCSE Award Lecture (pp. 1398-1404). Comm. of the ACM 31(12)
Faltin, N., (2001). *Structure and Constrains in Interactive Exploratory Algorithm Learning*. Springer LNCS issue on Software Visualization. S. Diehl (Ed.): Software Visualization International Seminar, Dagstuhl Castle, Germany. Revised Papers LNCS 2269, (pp. 213-226).
Feng, Zhe (2003). Algorithm Visualization: Using Various Levels of Abstraction and Invariants. Honours Thesis, Acadia University.
Fleischer, R., & Kucera, L. (2001). *Algorithm Animation for Teaching*. Springer LNCS issue on Software Visualization. S. Diehl (Ed.): Software Visualization International Seminar, Dagstuhl Castle, Germany, Revised Papers LNCS 2269, (pp. 113-128).
Fowler, M. (2000). *UML Distilled*. (Second Edition). Addison-Wesley, Reading, Mass.
Gamma, E., & Helm, R. & Johnson, J., & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.
Gloor, P. A., (1992). AACE - Algorithm Animation for Computer Science Education. In Proceedings of the *1992 IEEE Workshop on Visual Languages*, Seattle, WA (pp. 25–31).

Gloor, P. A., (1998). Animated Algorithms. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 409-416). MIT Press.
Goodrich, M. and Tamassia, R., (2001). *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., 2nd edition.
Horstmann, C. (2001) *Big Java: Programming and Practice*. John Wiley & Sons.
Hundhausen, C.D., Douglas, S.A., & Stasko, J.T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing 13(3)*, (pp. 259-290).
Müldner, T. and Shakshuki, E. (2004a), “A New Approach to Learning Algorithms”, submitted to the International Conference on Information Technology: Coding and Computing, April, 2004.
Müldner, T. and Shakshuki, E. (2004b), “Teaching Students to Implement Algorithms”, submitted to the 9th Annual Conference on Innovation and Technology in Computer Science Education; ITiCSE 2004.
Petre, M., Baecker, R, & Small, I. (1998a). An Introduction to Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 3-26). MIT Press.
Petre, M., Blackwell, A. F., & Green, T. R. G. (1998b). Cognitive Questions in Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 453-480). MIT Press.
Price, B., Baecker, R. & Small, I. (1998). An Introduction to Software Visualization. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 7-27.). MIT Press.
Stasko, J., & Lawrence, A. (1998). Empirically Assessing Algorithm Animations as Learning Tools. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price (Eds.), *Software Visualization: Programming as a Multi-Media Experience* (pp. 417-419). MIT Press.
Stern, L., & Naish, L. (2002a). Animating Recursive Algorithms. *Interactive Multimedia Electronic Journal of Computer Enhanced Learning*, Volume 4, Number 2.
Stern, L., & Naish, L. (2002b). *Visual Representation for Recursive* Algorithms. In Proceedings of the 33rd Annual SIGCSE Technical Symposium on Computer Science Education, Association for Computing Machinery (pp. 196-200).
Whitley, J., & King, A. (2001). Animation of insertion sort. http://www.cs.duke.edu/courses/spring01/cps049s/students/ack11/jawaasorting.html
Wilhelm, R., Müldner, T. & Seidel, R. (2001). Algorithm Explanation: Visualizing Abstract States and Invariants. Springer LNCS issue on Software Visualization. S. Diehl (Ed.): Software Visualization International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. *Revised Papers LNCS 2269*, (pp. 381-394).