A C++ App for Demonstration of Sorting Algorithms on Mobile Platforms

http://dx.doi.org/10.3991/ijim.v8i1.3464

R. Meolic and T. Dogša
University of Maribor, Maribor Slovenia

***Abstract*—This paper presents a systematic approach for designing a C++ app for demonstrating sorting algorithms on smartphones and tablets. It is part of an on-going project on the usages of new technologies in education. The general properties of mobile platforms are discussed together with details specific to demonstrating sorting algorithms. The implementation of Insertion Sort is given as an example. The obtained results (e.g. design rules and algorithms) have been tested within a prototype application.**

***Index Terms* — Usability, Qt, Engineering Education, Mobile learning**

I. INTRODUCTION

The American Dialect Society named "app" the word of the year for 2010. Though it is a short for "application" it is rarely used to denote just any computer program but only those intended to run on a mobile platform i.e. on smartphones, tablets, and similar devices. Apps are also special in their methods of distribution. Authors upload and users download them from special web-stores called markets. This is the most effective way of software distribution ever. Just anyone can publish or sell an app. From amongst millions of these apps, many of them are oriented towards mobile learning. It is expected that mobile platforms will influence the education process at least as much as personal computers and the internet. The benefits of mobile learning include learning from rich interactive content (in contrast to books), flexible learning locations (in contrast to PCs), and at flexible learning times (in contrast to classrooms).

The C++ programming language was born in the early 1980s and it was first standardised in 1998. The current standard was ratified in 2011. Although C++ itself is widespread and ubiquitous, C++ apps are not. Default system development kits (SDK) for today's more popular mobile platforms do not emphasise C++. Apps for Apple's iOS are written in Objective-C, Microsoft's Windows Phone apps are created with C#, and Google's Android apps are Java programs. On the other hand, all these platforms allow for the developing of C++ apps, iOS by simply integrating different programming technologies together (e.g. Objective-C++) and Windows Phone and Android by provided support for (ambiguously named) native code. In a standard model–view–controller (MVC) design, the "model" and "controller" parts which consist of data and functions for data manipulation are suitable targets for C++ implementation, whilst the "view" part which is responsible for data representation and user interface is usually bound to the default technology. When considering the broader spectrum of mobile platforms, the situation is more challenging. For many of them, C++ apps are first class citizens, i.e. C++ is used thoroughly.

Sorting is a relatively narrow and straightforward subject as long as we are not creating science from it [1]. In [2] the author states that sorting is the fundamental algorithmic problem in computer science and that learning the different sorting algorithms is like learning scales for a musician. Indeed, sorting is the first step in solving a host of other algorithmic problems. Today, the pseudocodes of different sorting algorithms and their implementations in different programming languages can easily be found on the internet (e.g. project Rosetta Code). However, the most respected reference is the 3rd volume of Knuth's encyclopaedia [3].

Engineers do not learn computer algorithms only for being able to write their straightforward implementation — efficient implementations are already present in libraries. The goal of knowing various algorithms is to be capable of adapting and extending existing ones and to help in inventing completely new algorithms. Indeed, one can benefit a lot from studying a high quality algorithm's implementation. Graphical debuggers allow for easy observation of every memory bit, and the tracking of any program flow. But these tools are really oriented towards testing and debugging, not teaching. A purpose-built demonstration of an algorithm may differ considerably from the debugger approach:

- Pseudocode can be used instead of a real programming language;
- Program flow can be reversed;
- Actions can be explained in advance;
- Explanations can go beyond simple comments;
- Additional functionality can be added such as an examination mode for testing the user's knowledge.

When overviewing the markets we found apps dedicated to sorting algorithms that already include some of these functionalities, e.g. the app described in [4] encourages learning by awarding students with points. However, existing apps are still pretty elementary, the ultimate app (will there ever be one?) should include a large set of functionalities already available in Java and JavaScript-based applications on the Web. Let us mention some of them that could serve as the origins of ideas.

The well-known web site called Sorting Algorithm Animations (http://www.sorting-algorithms.com/) gives a plain visual comparison between different sorting algorithms and their efficiencies. The main properties and very formal pseudocode are also given for each algorithm.

Project JHAVE includes demonstrations of different sorting algorithms (http://jhave.org/). It is a client-server project where each demonstration is implemented in a special scripting language. The demonstrations look different from each other because they are precisely created to explain each particular algorithm.

D. K. Nester set up an interesting web page, that shows pseudocode for various sorting algorithms and enables the tracing of their executions (http://www.bluffton.edu/~nesterd/java/SortingDemo.html).

This paper is an updated report on the on-going project as recently presented at the International Conference on Computer Supported Education [5]. It is further organised as follows. In Section II, the design of an universal framework is discussed. Section III provides specific details of tracking sorting algorithms. In Section IV, a prototype app is described.

II. AN UNIVERSAL MOBILE FRAMEWORK FOR DEMONSTRATION OF SORTING ALGORITHMS

In order to help developers create consistent apps, vendors provide default application frameworks for their platforms and may also offer a dedicated development environments (IDE). In contrast, there is a tendency to use third-party application frameworks which supports cross-platform development. Surely, multi-platform support can be the more easily addressed with web technologies but such solutions are not always suitable or flexible enough [6]. In the presented project we aim for portability but we also want to support complex data structures that cannot be efficiently implemented using scripting languages. We have chosen Qt (http://qt.digia.com/), a default application framework for BlackBerry 10, Sailfish OS, and Ubuntu Touch, which is actively being extended to also become also a third-party framework for Android and iOS. Qt supports all standard libraries and adds a special signals and slots mechanism for communication between objects. Effective user interfaces for Qt-based apps are designed for using the declarative language QML. Qt Creator (Fig. 1) is very helpful while developing Qt-based apps.

Mobile platforms share many common properties such as touchscreen display, virtual keyboard on demand, and various sensors. However, developing apps that can be used on different platforms is often extremely difficult [7]. For example, when taking into account the limited screen size and resolution of an average device it is important to create a minimalistic GUI. We found a helpful review of a mobile interface design in [8]. Based on the feedback from users, it is suggested that:

- All description texts that do not provide extra knowledge should be omitted;
- The appropriate colours and text sizes are very important for increasing readability and clarity;
- The usages of icons and images should be limited;
- Concepts from Web applications may help users to transfer known user experience to the mobile app, but input methods and menu organisation should resemble standard mobile functionalities.

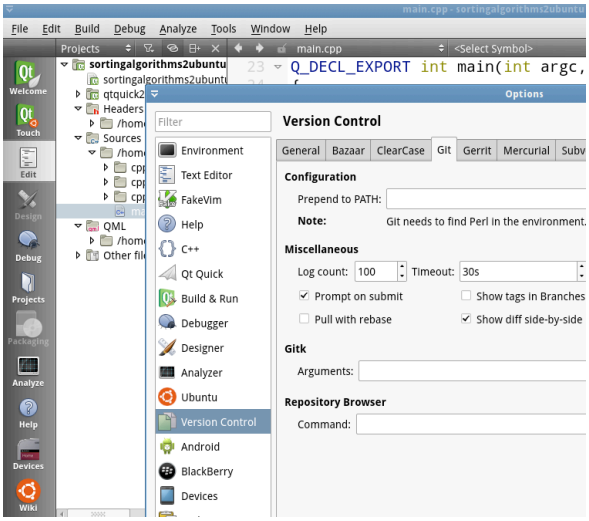

Figure 1. Qt Creator is a powerful IDE for cross-platform development

There is a lot of information, numerical and graphical, that we want to present to the user. However, we cannot show it all at once because of the limited space and because this would be too demanding for the user. Some possible solutions are:

- Let the device show different data in the portrait and landscape modes;
- Use pop-ups to show detailed data about items;
- Implement different screens that can be navigated by using tabs or swipe gestures;
- Use a hierarchical presentation where different groups and subgroups of data can be shown (expanded) or hidden (collapsed).

There is also a wide choice of mechanisms for controlling the app's run:

- Tapping and other gestures on the touchscreen;
- Rotating and other movements perceived by different sensors;
- Visual gestures observed by a camera;
- Voice commands, etc.

Let us discuss the advantage of orientation detection, only (Fig. 2). The portrait view is an obvious choice to show source code. Thus, the greater part of the screen may be reserved for that component. Thus the table of elements must be as small as possible but readable. Before an algorithm is chosen, the area reserved for source code can be used to show statistics. The landscape view may serve

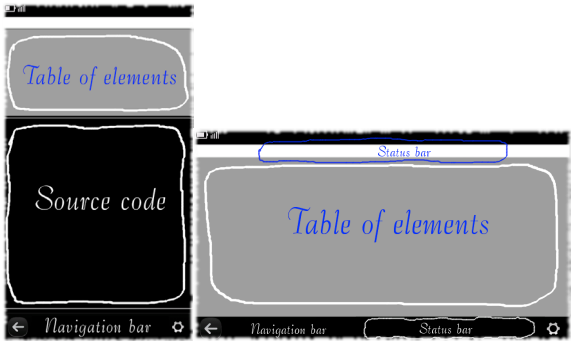

Figure 2. Portait and landscape view show different data

for presentations and interactions unrealised on the portrait view. Large elements make the user's interaction with them easier, thus almost all the area should be occupied by a table of elements. A status bar with one-line comment about the algorithm's current action could be added to allow for tracking the algorithm even without seeing it completely.

III. TRACKING SORTING ALGORITHMS ON MOBILE PLATFORMS

Sorting algorithms can be demonstrated in many different ways. In order to compare their effectiveness it is best to run them concurrently and use only global view (e.g. each element is represented by a bar). To emphasise the differences, animations with fancy effects (e.g. colours or sounds) can be created. In order to explain the implementation details, pseudocode may be involved.

Detailed tracking of the algorithm's run seems to be the crucial functionality for deep understanding. Due to limited space let us discuss this aspect only. We are interested in the basic form of sorting algorithms where the elements to be sorted are stored in the table and no optimisations are used (i.e. each time the algorithm needs a value of an element it performs a memory read). Our goal is using the same framework for demonstrating the various sorting algorithms. In order to display details, differing algorithms may require different approaches. However, in our approach we sacrifice some flexibility for uniformity because we obtain more direct comparisons between different solutions. Also, shared platforms for demonstrating the problems and solutions are well accepted in computer science teaching books.

In order to study an algorithm one has to view it in one form or another. We went for the pseudocode in the form of a simplified source code because this is concise representation whilst also suitable for the usage within a basic computer science course. Hence, each algorithm is composed of sentences that are either assignments, decisions, or flow control statements. We preferred C-style syntax for assignments, keywords, and parenthesis but we have discarded the semi-colons at the end of sentences (to reduce the text width and also to make it more appealing for developers using scripting languages). An indent of 2 spaces is the most we could afford on a small screen. Syntax highlighting is used to make pseudocode comprehensible. We omitted variable declarations but assumed that all the elements and indices are integers. The following naming scheme is used:

- The number of elements is denoted by `n`;
- The value of the i-th element is denoted by `[i]`, i.e. the name of the table is discarded;
- The loops use variables `i` and `j`, whereas other variables are named as usual in the literature.

As well as the syntax, we paid careful attention to the presentation of the semantics. The inclusion of compound and otherwise complicated sentences is undesirable. Flow control by using the "for" loop is a typical example of a complex statement that is composed of an initial assignment, a condition to stop, and control actions applied at the end of every looping. Thus, we preferred "while" or "foreach" loops over the "for" loop within the pseudocode.

In order to avoid an inconsistent handling of different semantic parts, we identify a set of atomic operations (i.e functions) that reveal the concept of sorting algorithms. Atomic operations are those that will create actions on the screen, either by simply showing/changing some values or applying some graphical effects, e.g. highlighting a part of the screen, doing some animation etc. Moreover, the atomic operations correspond to an algorithm's steps during step-by-step tracing. To some extent, the atomic operations coincide with the pseudocode statements, but we would have lost all the flexibility by equating these two formalisms. The following atomic operations are necessary and sufficient:

- Reading the value of an element;
- Changing the value of a variable;
- Comparing the values;
- Starting/continuing/finishing a loop;
- Swapping and moving elements.

We consider swapping and moving elements to be atomic operations. They are autonomous principles usually taught along with the sorting algorithms.

Once atomic operations are being identified, we can determine the necessary highlighting effects. Two mark-up functions are quite enough for a simple app:

- Function `setMark` that denote which element is currently observing;
- Function `setSpecial` that denote elements with some special feature, e.g. they are already in order.

Functions `setMark` and `setSpecial` can be used for the same element simultaneously and therefore their visual effects must be compatible. All other informations will be presented through labels and comments. Swapping and moving elements are supposed to be animated and hence we do not need extra effects for these operations.

The user studying sorting algorithms could be interested in different statistics about the efficiency, e.g. the number of required data comparisons, memory reads, memory writes, etc. Static bounding of statistical data to the pseudocode statements is adequate for many situations, e.g. each swap of two elements consists of 2 write operations (plus 2 read operations if the values are not already stored in local variables). But once again, such fixation is inflexible and, for example, in the case of compound decisions or in the case of single-line loops it is also very impractical. Thus we propose explicit manipulation of statistical data via special functions.

In order to illustrate the presented approach, let us observe the source code for demonstration of Insertion Sort given in Fig. 3. The bold lines correspond to the lines of pseudocode that is shown to the user. Function `CC` is used to identify and describe single steps. Functions `setLabel` and `incrLabel` are used to update the values of labels and statistical counters. A log from running the presented implementation is illustrated in Fig. 4. There, elements marked with functions `setMark` and `setSpecial` are illustrated as framed or shaded, respectively. For each step, the corresponding pseudocode line number and comment are displayed. Please note, that it is up to controller to implement step-by-step and backward running.

```
insertionSort(table T, int n) {
  int i, j, key;
  setLabel("i",-1);
  setLabel("j",-1);
  setLabel("key",-1);
  setLabel("read",0);
  setLabel("write",0);
  setLabel("sourceLine",1);
  setSpecial(0);
  CC("Skip first element");
  i = 1;
  while (i < n) {
    setLabel("sourceLine",2);
    setLabel("i",i);
    setMark(i);
    CC("Outer loop now at i");
    key = T[i];
    incrLabel("read",1);
    setLabel("key",key);
    setLabel("sourceLine",4);
    CC("[i] stored into variable key");
    j = i - 1;
    if (j >= 0) {
      setLabel("sourceLine",5);
      setLabel("j",j);
      setMark(j);
      CC("Inner loop starts at j=i-1");
      while (j >= 0 && T[j] > key) {
        incrLabel("read",1);
        setLabel("sourceLine",6);
        CC("Decrement j ([j]>key)");
        clearMark(j);
        j--;
        setLabel("j",j);
        if (j >= 0) setMark(j);
      }
    }
    setSourceLine(7);
    if (j >= 0) {
      incrLabel("read",1);
      CC("Stop inner loop ([j]<=key)");
      clearMark(j);
    } else \{
      CC("Stop inner loop (j<0)");
    }
    if (j != i-1) {
      setLabel("sourceLine",8);
      CC("Move [i] to index j+1");
      clearMark(i);
      move(i,j+1);
      incrLabel("write",i-j);
    }
    if (j == i-1) clearMark(i);
    setSpecial(j+1);
    setLabel("sourceLine",9);
    if (j >= 0) setLabel("j",-1);
    CC("Increment i");
    i++;
    setLabel("key",-1);
  }
  setLabel("sourceLine",11);
  setLabel("i",i);
  CC("All elements sorted");
}
```

Figure 3. Implementation of Insertion Sort

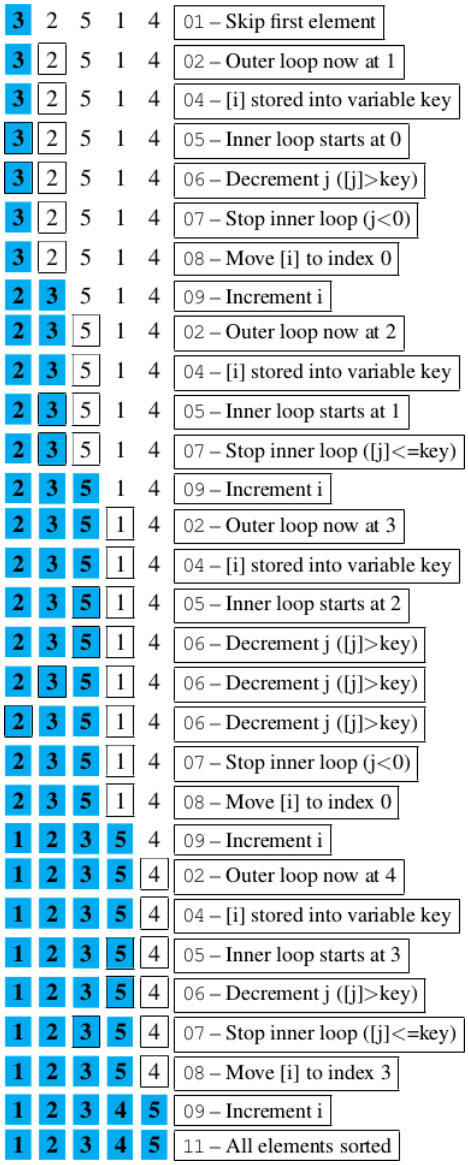

Figure 4. The log from running the Insertion Sort

IV. THE PROTOTYPE APP

We have created a prototype app for Nokia N9. This smartphone uses Meego OS and allows native C++ apps (devices with more recent Sailfish OS or Ubuntu Touch are unavailable as yet and Qt for Android is still a kind of beta software). The prototype app is planned to have Teaching mode (demonstrating and teaching), Practice mode (checking the user's knowledge), and Game mode (learning by playing). In version 0.2, only a part of Teaching mode is implemented with about 2,300 lines of C++ code and about 1,800 lines of QML code. Gnome sort, Insertion Sort, and Quicksort are included. Fig. 5 shows that the structure of the C++ code is carefully designed to separate the "model" part (a pure C++ code) from the "controller" part (a Qt specific code). Fig. 6 and Fig. 7 give some screenshots from prototype app.

Furthermore, we briefly discuss the implemented algorithms. Please, compare the given pseudocodes with the implementations, as given in Fig. 3 and [5].

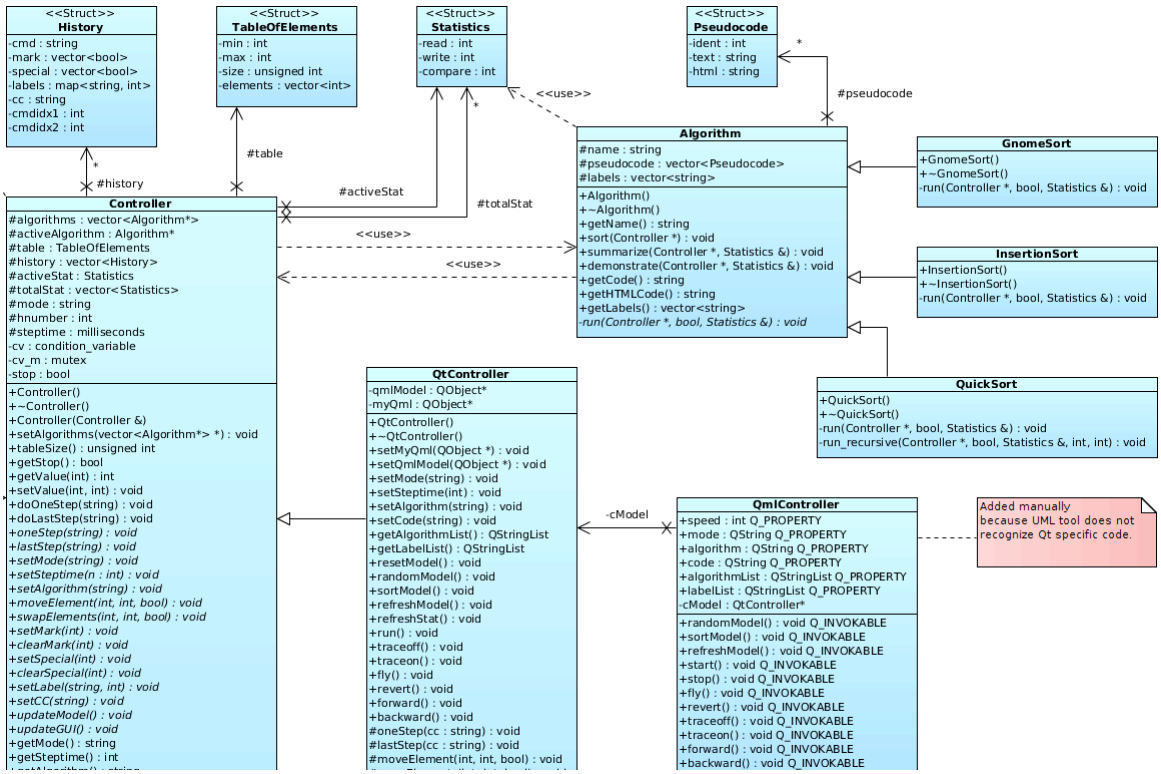

Figure 5. Class diagram showing a C++ part of prototype app

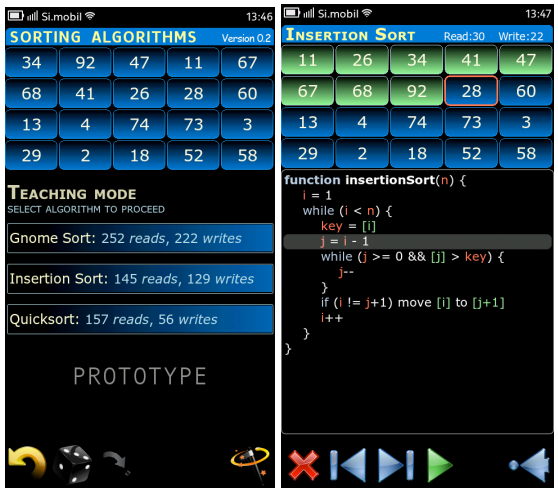

Figure 6. When algorithm is not selected, app shows the list of implemented algorithms together with short statistic data; when the algorithm is running, app shows the actual table of elements and pseudocode with marked current line

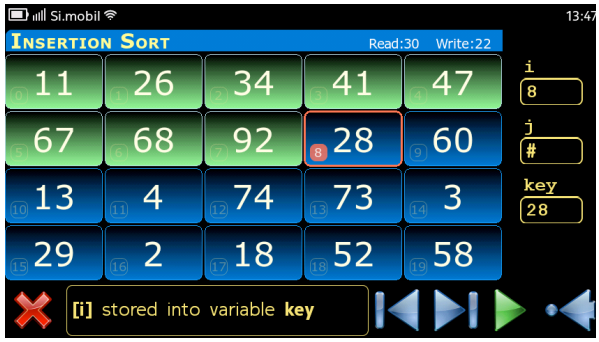

Figure 7. In landscape mode, app shows actual table of elements, the value of labels and the comment for current action

A. *Gnome Sort*

Gnome Sort is advertised as the technique used by the standard Dutch garden gnome to sort a line of flower pots. Gnome Sort is supposed to be the simplest sorting algorithm and it is indeed very easy to demonstrate it. The pseudocode of the Gnome Sort as shown in the prototype app is given in Fig. 9. The index used by the algorithm (yes, Gnome Sort uses only one index) is called "position" and is for brevity denoted by variable `i`. We use function `setMark` to mark the element on the index "position" and function `setSpecial` to denote the elements that are already in order.

B. *Insertion Sort*

Insertion Sort (Fig. 10) is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large arrays than more advanced algorithms, but for small arrays it can even outperform them. Moreover, Insertion sort handles nearly sorted arrays quite efficiently. When humans manually sort something (e.g. a deck of playing cards), most of them use a method similar to Insertion Sort. In our implementation of Insertion Sort we use function `setMark` to denote elements with indices `i` and `j`, that are currently being compared, and function `setSpecial` to mark the already sorted elements (see Fig. 3 and Fig. 4).

C. *Quicksort*

Quicksort is a divide-and-conquer algorithm that first divides the elements into two subgroups and then subsequently sorts these subgroups. Many versions exist of the original algorithm, which are either called a simple version or an in-place version. In our project we have

chosen one of the in-place versions (Fig. 11), i.e. an algorithm that does not need an extra space for dividing elements into two subgroups. Moreover, we used a recursive version of the algorithm, which is also the more common. The presented variant is not the original version of the algorithm [9] but in our opinion it is the more appropriate one for the use in the class. Please, note that we omitted details of pivot calculation to make pseudocode shorter. In the prototype app we used the median of the first, middle, and last elements.

```
 0: function gnomeSort(n) {
 1:   i = 0
 2:   while (i < n) {
 3:     if (i == 0 || [i] >= [i-1]) {
 4:       i++
 5:     } else {
 6:       swap [i] and [i-1]
 7:       i--
 8:     }
 9:   }
10: }
```

Figure 8. Pseudocode of Gnome Sort

```
 0: function insertionSort(n) {
 1:   i = 1
 2:   while (i < n) {
 3:     key = [i]
 4:     j = i-1
 5:     while (j >= 0 && [j] > key) {
 6:       j--
 7:     }
 8:     if (j != i-1) move [i] to [j+1]
 9:     i++
10:   }
11: }
```

Figure 9. Pseudocode of Insertion Sort

```
 0: function quickSort(begin, end) {
 1:   left = begin
 2:   right = end
 3:   if (left < right) {
 4:     calculate pivot
 5:     while (left <= right) {
 6:       while ([left] < pivot) left++
 7:       while ([right] > pivot) right--
 8:       if (left <= right) {
 9:         swap [left] and [right]
10:         left++
11:         right--
12:       }
13:     }
14:     quickSort(begin,right)
15:     quickSort(left,end)
16:   }
17: }
```

Figure 10. Pseudocode of Quicksort

V. CONCLUSION

We have studied sorting algorithms with the goal of tracking their run by using a set of given visual effects whilst taking into account the limitations of the mobile platform. It is important to choose the proper presentation of the algorithm's flow and the proper type and amount of visual effects. We successfully managed three different algorithms, so we believe that our approach can be extended for most of the other sorting algorithms as well.

The paper has reported about a prototype app for a specific mobile platform. Nevertheless, we have researched and discussed all the problems in such a general way that the results are applicable on any mobile platform. Each statement in C++ code either belongs to the sorting algorithm, or collects statistical data, or serves as a part of the demonstration strategy. We were careful not to mix these roles. Hence, the obtained code is very portable. Furthermore, the statistical data can be obtained by only excluding all GUI statements.

Although not being the main goal of this project, the obtained platform itself was found usable when teaching software engineering. Today, students often underestimate the benefit of appropriate planning (e.g. preparation of graphical views) and appropriate testing (e.g. preparation of test cases). Developing an app similar to the presented one is quite a challenge due to the limited screen sizes on mobile devices and because the effective demonstration of sorting algorithms involves many navigation options.

REFERENCES

[1] P. Vitanyi, "Analysis of Sorting Algorithms by Kolmogorov Complexity (a survey)". In *Entropy, Search, Complexity*, pp. 209–232, 2007. http://dx.doi.org/10.1007/978-3-540-32777-6_9

[2] S. S. Skiena, *The Algorithm Design Manual*. Springer, 2nd ed., 2008.

[3] D. E. Knuth, *The Art of Computer Programming*, 2nd ed., vol. 3: Sorting and Searching, Addison-Wesley Professional, 1998.

[4] I. Boticki, A. Barisic, S. Martin, and N. Drljevic, "Teaching and Learning Computer Science Sorting Algorithms with Mobile Devices: A case study," *Computer Applications in Engineering Education*, vol. 21, pp. E41-E50, 2013. http://dx.doi.org/10.1002/cae.21561

[5] R. Meolic, "Demonstration of Sorting Algorithms on Mobile Platforms," In *The Fifth International Conference on Computer Supported Education*, pp. 136-141, 2013.

[6] W. Jobe, "Native Apps vs. Mobile Web Apps", *International Journal of Interactive Mobile Technologies*, vol. 7, no. 4, 2013.

[7] A. Holzinger, P. Treitler, and W. Slany, "Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones," In *Multidisciplinary Research and Practice for Information Systems*, volume 7465 of LNCS, pp. 176–189, 2012.

[8] J. Mirkovic, H. Bryhni, and C. M. Ruland, "Designing User Friendly Mobile Application to Assist Cancer Patients in Illness Management". In *The Third International Conference on eHealth, Telemedicine, and Social Medicine*, pp. 64–71, 2011.

[9] L. Khreisat, "Quicksort — a Historical Perspective and Empirical Study," *International Journal of Computer Science and Network Security*, 2007, vol. 7, no.12, pp. 54–65, 2007.

AUTHORS

R. Meolic is with the Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, SI-2000 Slovenia. (e-mail: meolic@uni-mb.si).

T. Dogša is with the Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, SI-2000 Slovenia. (e-mail: tdogsa@uni-mb.si).

This article is an extended and modified version of a paper presented at the 5th International Conference on Computer Supported Education (CSEDU 2013), 6-8 May 2013, Aachen, Germany. Submitted 06 December 2013. Published as re-submitted by the authrs 05 January 2014.