Demonstration of Sorting Algorithms on Mobile Platforms

Robert Meolic
Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia
meolic@uni-mb.si

This is an extended version of the paper published in the proceedings of CSEDU 2013, pp. 136-141, 2013.

Keywords: Learning Software, Mobile Learning, Computer Science, Engineering Education

Abstract: This paper presents a systematic approach to the implementation of a mobile framework for demonstrating of sorting algorithms. Well-known corresponding projects are listed. The pseudo-code form is discussed, together with the graphical presentation of actions. A set of atomic operations that reveal the concept of sorting algorithms has been identified. Implementations of Gnome Sort, Insertion Sort, and Quicksort are given as a portable C-style code. The presented code has been tested within a prototype application on a modern smartphone. The project was oriented towards the usage in mobile learning systems.

1 INTRODUCTION

There are many subjects and concepts within computer science. Some of them are broad and extensive whilst others are relatively narrow and specific. Sorting algorithms can be classified into the last group as long as we are not creating science from them (Vitanyi, 2007). In (Skiena, 2008) 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. Pseudo-codes of different sorting algorithms and their implementation in different programming languages can easily be found on the internet (Wikipedia, 2012), (Rosetta Code, 2012). However, the most respected reference is the 3rd volume of Knuth's encyclopedia (Knuth, 1998).

The number of mobile platforms has been increasing rapidly, and it is expected that they 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 and at flexible learning times (in contrast to classrooms). Smartphones, that are becoming the more widespread mobile platforms, are introducing another interesting factor, the application market, which is the most effective way of software distribution ever. Just anyone can publish his/her application. From amongst millions of these applications, many of them are already oriented towards mobile learning.

We are definitely not the first nor alone in researching the usage of mobile applications for teaching various topics regarding computer programming. We have been directly encouraged by a recent paper on teaching sorting algorithms using an Android-based application (Boticki et al., 2012). It encourages learning by location-independent usability and by awarding students with points.

Many applications on mobile platforms resemble Java applets and other web applications. This is becoming even more frequent as new ubiquitous web technologies (e.g. HTML 5) can be directly used to develop applications on mobile platforms. Let us mention some existing web applications visualising sorting algorithms.

The web site called Sorting Algorithm Animations (Martin, 2012) provides an interesting comparison of the algorithms' efficiencies. The main properties and very formal pseudo-codes are given for each algorithm. Please, note that our goal is not the same, e.g. we do not want to only provide a plain description or comparison of different sorting algorithms.

Project JHAVÉ (Naps et al., 2012) includes different sorting algorithms. It is a client-server project where each demonstration is implemented in a special scripting language. In principle, the demonstrations of each algorithm looks differently because they are precisely created to explain each particular algorithm.

D. K. Nester set up a web page powered by Javascript, that shows the pseudo-codes for various sorting algorithms and enables the tracing of their executions (Nester, 2012). The outline of our work is very similar but we have involved mobile platforms.

2 MOBILE FRAMEWORK FOR DEMONSTRATING OF SORTING ALGORITHMS

A computer program is given by its source code. As a premise for a good design the code should be clear, difficult statements should be accompanied by explanatory comments. Moreover, graphical debuggers allow for easy observation of every memory bit, and the tracking of any program flow. However, designing the algorithm is not the same as implementing and testing it. Indeed, engineers do not learn computer algorithms only to be 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.

A purpose-built demonstration of an algorithm may differ considerably from the debugger approach:

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

A wide-selection of mobile platforms already exists, and this technology is still expanding very quickly. Smartphones, handhelds, gaming consoles, tablets, and ultrabooks, all of them are mobile in the sense of portability. Developing applications that can be used on different platforms is often extremely difficult (Holzinger et al., 2012). In order to address at least a part of this problem we were interested in a rather minimal GUI, and set only a few expectations:

- A display with a least 800x480 points;
- A touchscreen display recognising click, click-and-hold, and drag;
- A sensor for detecting screen orientation;
- A virtual keyboard on demand.

We found a helpful review of a mobile interface design in (Mirkovic et al., 2011). Based on the feedback from users, they suggested that:

- All description texts that do not give extra knowledge should be omitted;
- The right colours and text sizes are very important for increasing readability and clarity;
- The usage of icons and images should be highly limited;
- Concepts from web applications helps users to transfer known user experience to the mobile application, but text input and menu organisation should resemble standard mobile functionalities.

There is a lot of information, numerical and graphical, that we want to present to the user. However, we cannot show it all in once because of the limited screen size and because this would be to 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 which can be navigated by using tabs or swipe gestures;
- Use a hierarchical presentation where different groups and/or subgroups of data can be shown (expanded) or hidden (collapsed).

We have decided to take advantage of orientation detection. The portrait view is an obvious choice to show source code. Thus, we give the largest part of the screen to that component. However, without observing data changes the user will not benefit much from watching the run of a source code. Thus we also show the table of elements as small as possible but readable. Before an algorithm is chosen, the area reserved for the source code is used to show statistics.

The landscape view serves for those presentations and interactions not realised on the portrait view. Almost all the area is occupied by a table of elements. Large elements make the user's interaction with them easier. We have added a one-line comment about the algorithm's next action. This allows for tracking the algorithm even without seeing it completely.

3 TRACKING THE EXECUTION OF SORTING ALGORITHMS

The goal was the tracking of sorting algorithms by using a set of given visual effects whilst taking into account the limitations of the target platform. Different algorithms require different approaches.

The initial decision went to the pseudo-code style used for the presentation of the algorithms. Since we aimed at the usage within a basic computer science course, the obvious choice was a procedural pseudo-code form. Hence, the sorting algorithms are composed of sentences that are either assignments, decisions, or flow control statements. As well as the syntax, we also paid careful attention to the presentation of the semantics. The inclusion of compound and otherwise complicated sentences is undesirable.

The flow control by using the `for` loop is a typical example of complex statement which is composed of an initial assignment, a condition to stop, and control actions applied at the end of every looping. Thus, we preferred `while` loops within the presentation.

In order to avoid an inconsistent handling of different semantic parts, we identify a set of atomic operations that reveal the concept of sorting algorithms. Atomic operations are those that will be emphasised on the screen, either simply by showing/changing some value or applying some graphical effect, e.g. highlighting part of the screen or doing some animation. Moreover, the atomic operations formed separate steps during step-by-step tracing. To some extent, the atomic operations correspond to the pseudo-code statements, but we would have lost all the flexibility by equating these two formalisms. The following atomic operations are necessary and sufficient:

- Changing the value of a variable;
- Starting/continuing/finishing a loop;
- Reading the value of an element;
- Comparing the value of one element with the value of another one or with some stored value;
- Swapping two elements;
- Moving elements.

We were interested in the basic form of sorting algorithms where the elements to be sorted were stored in the table and no fancy optimisations were used. We considered swapping and moving elements to be atomic operations. They were autonomous principles usually taught along the sorting algorithms.

3.1 Gnome Sort

Gnome Sort is advertised as the technique used by the standard Dutch garden gnome to sort a line of flower pots (Grune, 2012). Gnome Sort is supposed to be the simplest sorting algorithm and, indeed, it is very easy to demonstrate it. Our implementation of the algorithm is shown in Figure 1. Only the bold lines are shown to the user of the mobile application. The shown lines are slightly modified (to reduce the text width and also to make it more appealing for developers using different programming languages).

The index used by the algorithm (yes, Gnome Sort uses only one index) is for brevity denoted by variable `i` in the pseudo-code. We use function `setMark()` to mark the element on the index `i` and function `setSpecial()` to denote the elements that are already in order. We use function `CC()` to enable step-by-step tracing and to describe the current/next step.

3.2 Insertion Sort

Insertion Sort 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 (Figure 2) we use mark-up function `setMark()` to denote elements with indices `i` and `j`, that are currently being compared, and function `setSpecial()` to mark the already sorted elements. Please, note that we omitted the details of moving the elements. Moreover, to find a proper place for a current element we do not assume the moving of each individual element. We only perform the necessary comparisons and subsequently move a whole block of elements at once. Whilst such an approach is not used in most practical implementations of Insertion Sort (at least not in those sorting a table of elements), this divergence neither change the main idea of the algorithm nor causes troubles or misunderstandings.

3.3 Quicksort

Quicksort is a divide-and-conquer algorithm which first divides elements into two subgroups and then recursively 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 choose one of the in-place versions (Figure 3), i.e. an algorithm which 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 most common. The presented variant is not the original version of the algorithm (Khreisat, 2007) but in our opinion it is the most appropriate one for the use in the class.

At the beginning of each recursion pass, the indices of lower and upper elements are stored into variables `left` and `right` which are then increased and decreased, respectively. Each recursion pass ends when index `left` becomes greater or equal to index `right`. Please, note that we omitted details for function `calculatePivot()` to make the code shorter. In the prototype application we used the median of the first, middle and last elements. For the sake of simplicity, the given implementation detects the final condition at the beginning of each recursive call (we could make this decision at the end, just before each recursive call).

```
gnomeSort(table T, int n) {
  int i, prev=-1;
  setLabel("read",0);
  setLabel("write",0);
  setLabel("sourceLine",1);
  i = 0;
  setLabel("i",0);
  setMark(0);
  CC("Position starts at 0");
  while (i < count) {
    if (prev != -1) {
      setLabel("sourceLine",3);
      setLabel("i",i);
      clearMark(prev);
      setMark(i);
    }
    prev = i;
    CC("Position now at i");
    if (i == 0 || T[i] >= T[i-1]) {
      setLabel("sourceLine",4);
      if (i == 0) {
        CC("Increment position (i==0)");
      } else {
        incrLabel("read",2);
        CC("Increment position ([i]>=[i-1])");
      }
      setSpecial(i);
      i++;
    } else {
      setLabel("sourceLine",6);
      CC("Swap elements [i] and [i-1]");
      clearMark(i);
      swap(i,i-1);
      incrLabel("write",2);
      setLabel("sourceLine",7);
      CC("Decrement position");
      i--;
    }
  }
  setLabel("sourceLine",9);
  setLabel("i",i);
  clearMark(i-1);
  CC("Algorithm finished");
}
```

Figure 1: Implementation of Gnome Sort algorithm.

```
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 <b>i</b>");
    i++;
    setLabel("key",-1);
  }
  setLabel("sourceLine",11);
  setLabel("i",i);
  CC("Algorithm finished");
}
```

Figure 2: Implementation of Insertion Sort algorithm.

```
quickSort(table T, int begin, int end) {
  int left, right;
  int pivot;
  left = begin;
  right = end;
  if (left < right) {
    setLabel("sourceLine",3);
    setLabel("left",left);
    setLabel("right",right);
    setLabel("pivot",-1);
    setSpecial(left,right);
    CC("Sort elements from left to right");
    setLabel("sourceLine",4);
    CC("Calculate pivot");
    pivot = calculatePivot(left,right);
    incrLabel("read",3);
    setLabel("pivot",pivot);
    setLabel("sourceLine",5);
    CC("Start partitioning");
    while (left <= right) {
      setLabel("sourceLine",6);
      CC("Search forward from left");
      setMark(left);
      while (T[left] < pivot) {
        incrLabel("read",1);
        CC("Element less than pivot");
        clearMark(left);
        clearSpecial(left);
        left++;
        setLabel("left",left);
        setMark(left);
      }
      incrLabel("read",1);
      CC("Element not less than pivot");
      setLabel("sourceLine",7);
      CC("Search backward from right");
      setMark(right);
      clearSpecial(right);
      while (T[right] > pivot) {
        incrLabel("read",1);
        CC("Element greater than pivot");
        setSpecial(right);
        right--;
        setLabel("right",right);
        setMark(right);
        clearSpecial(right);
      }
      incrLabel("read",1);
      CC("Element not greater than pivot");
      clearMark(right);
      if (left <= right) {
        setLabel("sourceLine",9);
        clearMark(left);
        setSpecial(right);
        CC("Swap elements [left] and [right]");
        swap(left,right);
        incrLabel("write",2);
        setMark(right);
        clearSpecial(left);
        left++;
        right--;
        setLabel("left",left);
        setLabel("right",right);
      }
    }
    setLabel("sourceLine",12);
    CC("left greater than right");
    setLabel("left",-1);
    setLabel("right",-1);
    setMark(begin,right);
    CC("Elements sorted per pivot");
    clearMark(begin,end);
    clearSpecial(left,end);
    quickSort(begin,right);
    quickSort(left,end);
  }
  CC("Return from recursive call");
}
```

Figure 3: Implementation of Quicksort algorithm.

4 DISCUSSION AND CONCLUSION

We have studied sorting algorithms with the goal of their demonstration on mobile platforms. There are many aspects to such work, e.g. choosing the proper presentation of the pseudo-code and choosing the proper type and amount of visual effects. Because we successfully managed three very different algorithms, we believe that our approach can be smoothly extended to the most of other sorting algorithms as well. Indeed, we do have some scruples about using the same framework for demonstrating various sorting algorithms. It could, however, be acceptable to sacrifice some flexibility for uniformity because we obtained a more direct comparison between different solutions. All in all, the shared platforms for demonstrating the problems and solutions are quite common in computer science teaching books.

When considering the practical part of the project, the main results were the implementations of sorting algorithms. When they are observed more closely, it is noticeable that each statement either belongs to the sorting algorithm, or collects statistical data, or serves as a part of the demonstration strategy. We were very careful not to mix these roles. Hence, the obtained C-style code is portable and not bound to any specific graphical API. Furthermore, we can obtain statistical data by only excluding all GUI statements.

The presented implementations were tested in a prototype application on Nokia N9. This smartphone uses Meego OS (based on Linux kernel) and allows native applications in C++. Figures 4 and 5 give some screenshots from the prototype application. In any case, the goal of the project was to research the problems in such a general way that the implementation on any mobile platform could benefit from it.

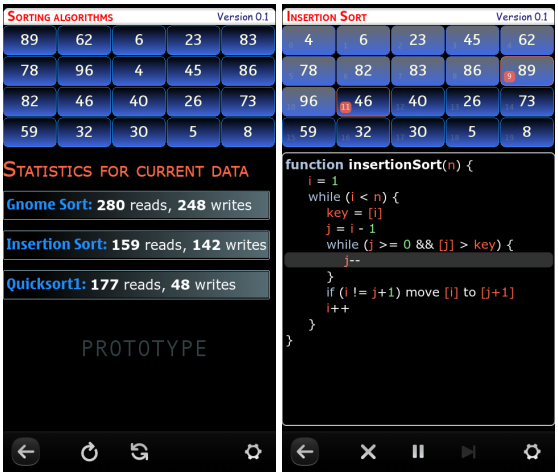

Figure 4: Prototype application running on Nokia N9.

Figure 5: Prototype application in landscape mode.

Although not being the main goal of this project, we found it very usable when teaching software project management. Today, students often underestimate the benefit of appropriate planning (e.g. preparation of graphical views) and testing (e.g. preparation of test cases). The presented project requires detailed planning of graphical representation because of the limited screen sizes on mobile devices, and because the effective demonstration of sorting algorithms involves several navigation commands (run, pause, stop, one step, reset, etc.), so testing the obtained application is quite a challenge.

Although, at least Quicksort algorithm is not trivial, sorting is not a real challenge for most of the students. This is not the case with all subjects. For example, when considering search trees, e.g. AVL trees, 2-3 trees, and red-black trees; have you ever been able to manage them? Every group of algorithms needs a special framework and more complex and longer algorithms are quite problematic for demonstrating on mobile platforms. Fortunately, mobile devices are becoming bigger and more powerful, capable of presenting more text and graphics, fancier animations, and effects. Also, many diverse controls are appearing (gestures, 3D accelerometers, voice commands). Hence, we all have a lot of further work.

REFERENCES

Boticki, I., Barisic, A., Martin, S., and Drljevic, N. (2012). Teaching and learning computer science sorting algorithms with mobile devices: A case study. *Computer Applications in Engineering Education*.

Grune, D. (2012). Gnome sort - the simplest sort algorithm. Retrieved December 14, 2012, from http://dickgrune.com/Programs/gnomesort.html.

Holzinger, A., Treitler, P., and Slany, W. (2012). 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 *Lecture Notes in Computer Science*, pages 176–189.

Khreisat, L. (2007). Quicksort — a historical perspective and empirical study. *International Journal of Computer Science and Network Security*, 7(12):54–65.

Knuth, D. E. (1998). *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley Professional, 2 edition.

Martin, D. R. (2012). Sorting algorithm animations. Retrieved December 14, 2012, from http://www.sorting-algorithms.com/.

Mirkovic, J., Bryhni, H., and Ruland, C. M. (2011). Designing user friendly mobile application to assist cancer patients in illness management. In *The Third International Conference on eHealth, Telemedicine, and Social Medicine*, pages 64–71.

Naps, T., Furcy, D., Grissom, S., and McNally, M. (2012). Java-hosted algorithm visualization environment. Retrieved December 14, 2012, from http://jhave.org/.

Nester, D. K. (2012). Sorting demonstrations. Retrieved December 14, 2012, from http://www.bluffton.edu/~nesterd/java/SortingDemo.html.

Rosetta Code (2012). Sorting algorithms. Retrieved December 14, 2012, from http://rosettacode.org/wiki/Category:Sorting_Algorithms.

Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer, 2 edition.

Vitanyi, P. (2007). Analysis of sorting algorithms by kolmogorov complexity (a survey). *Entropy, Search, Complexity*, pages 209–232.

Wikipedia (2012). Sorting algorithm. Retrieved December 14, 2012, from http://en.wikipedia.org/wiki/Sorting_algorithm.

APPENDIX

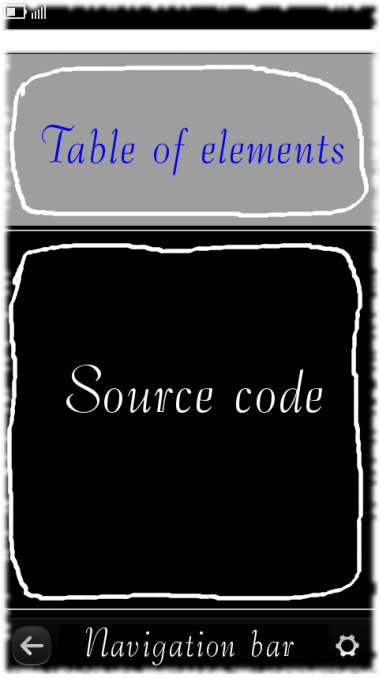

Figure 6: The outline of portrait view.

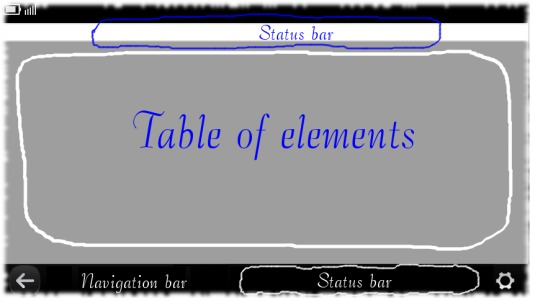

Figure 7: The outline of landscape view.

```
 0: 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: Gnome Sort algorithm as shown to the user.

```
 0: 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: Insertion Sort algorithm as shown to the user.

```
 0: 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: Quicksort algorithm as shown to the user.

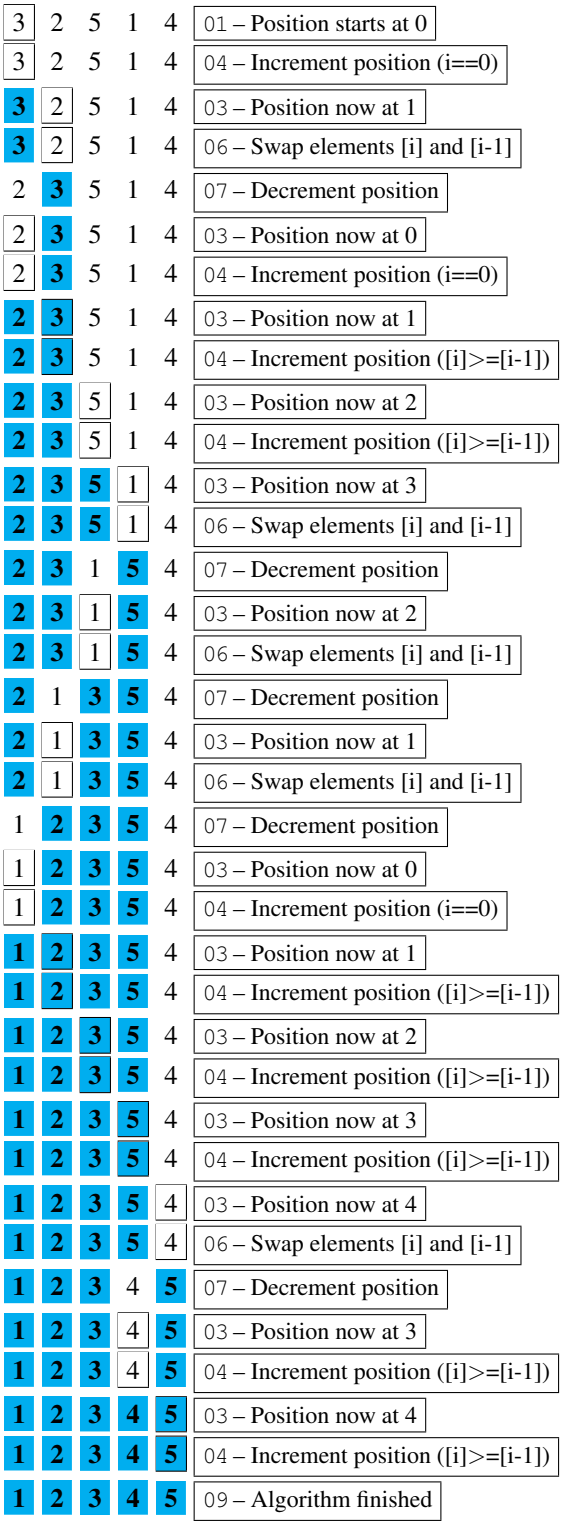

Figure 11: Demonstration of Gnome Sort

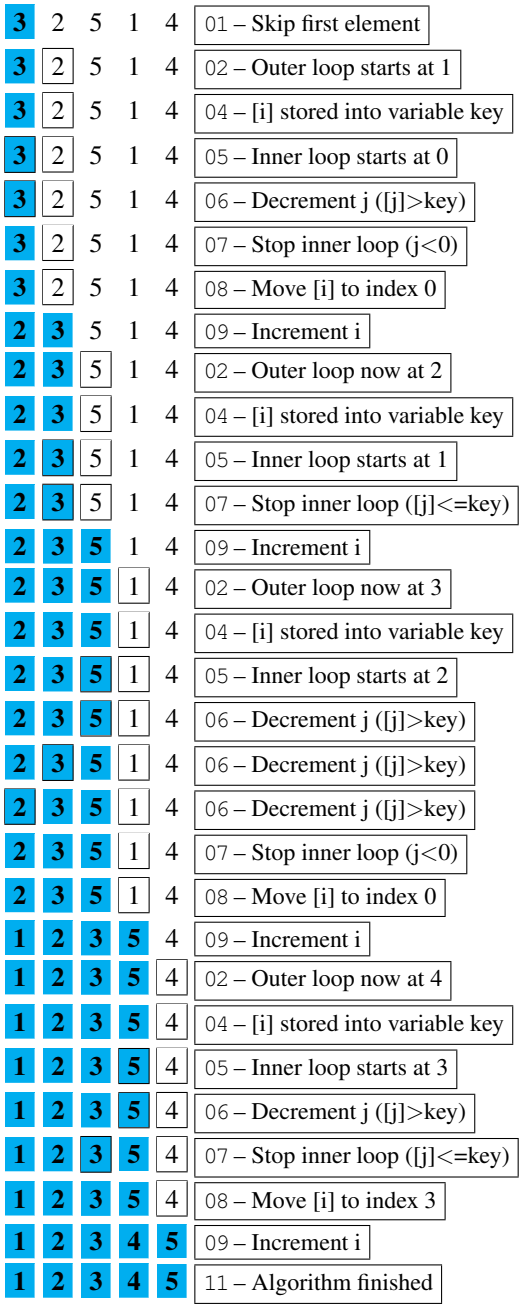

Figure 12: Demonstration of Insertion Sort.

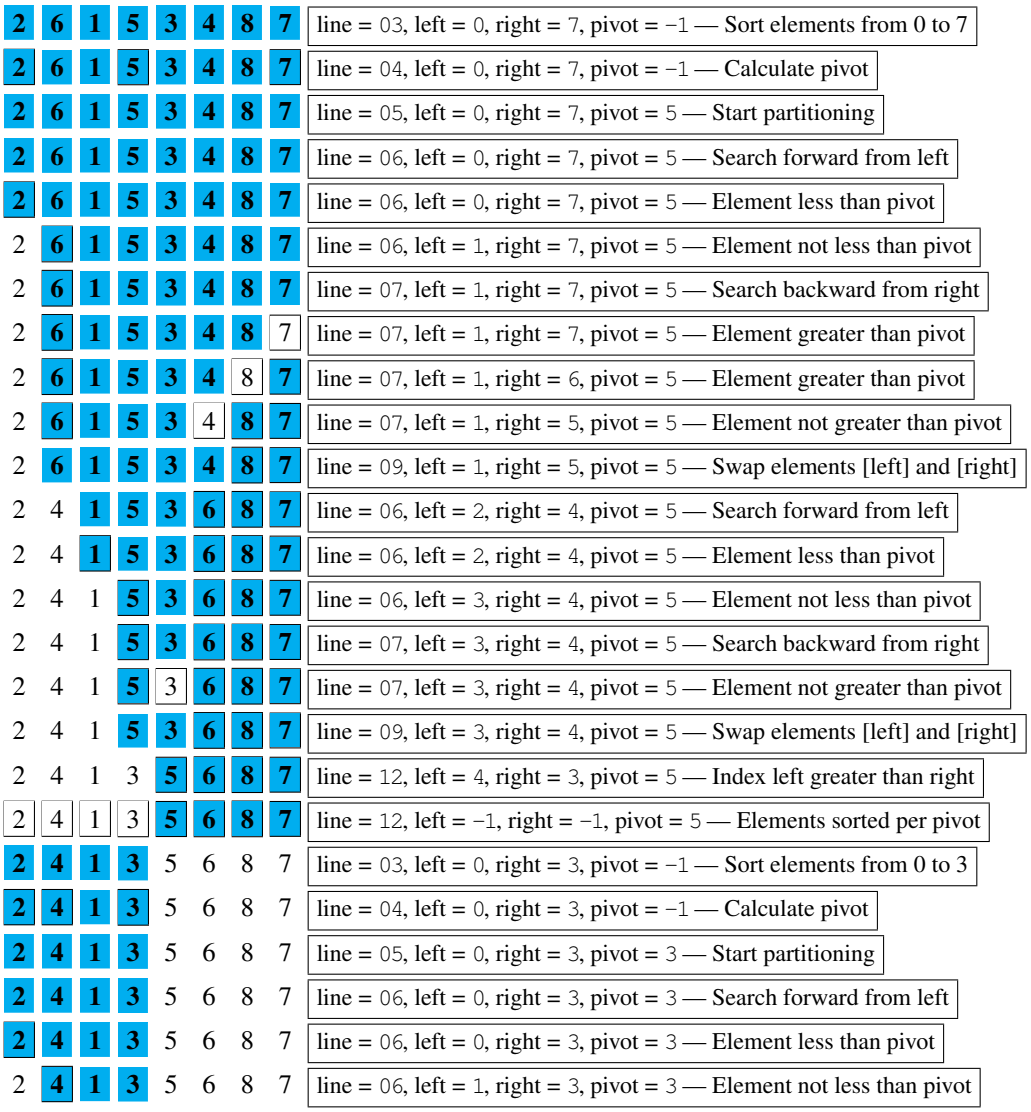

Figure 13: A part of Quicksort demonstration (`pivot` is the median of the first, middle and last elements).