Common sorting algorithms

Mikael Tylmad

February 27, 2008

1 Introduction

This report concerns some of the more common and easy to use sorting algorithms. More algorithms will be added as time progresses. The goal of the report is to explain the basic principles of each algorithm and to give an example of how the algorithm may be implemented. The programming language used is Java.

2 Selection sort

2.1 The algorithm

Selection sort is a very simple sorting algorithm. It works by following these steps:

1. Find the minimum value in the list by iterating over the whole list
2. Swap this value with the first value in the list
3. Repeat these two steps, but each time decrease the list by starting with the second position

By following these steps, the list is divided into two parts: a sorted part, the part to the left, and an unsorted part, the part to the right. With each iteration, the sorted part grows.

This algorithm is very simple to implement but has $\Theta(n^2)$ complexity which makes it inefficient on large lists. In certain situations however it has advantages. Since it always iterates over all the elements of the list i question, the sorting time can be pre-calculated. This could be valuable knowledge in some cases. Though this is also true with heap sort and quicksort which are both faster. Another advantage is that selection sort only performs $\Theta(n)$ swaps. This is preferable when using for example EEPROM or Flash where it is more expensive to write than to read.

2.2 Complexity analysis

Since selection sort always performs in the same way, regardless of the data, the complexity analysis becomes very simple. Each loop scans the entire list given. This takes $n-1$ comparisons the first time, $n-2$ comparisons the second time etc. This results in running time $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2 = \Theta(n^2)$.

2.3 Graphical example

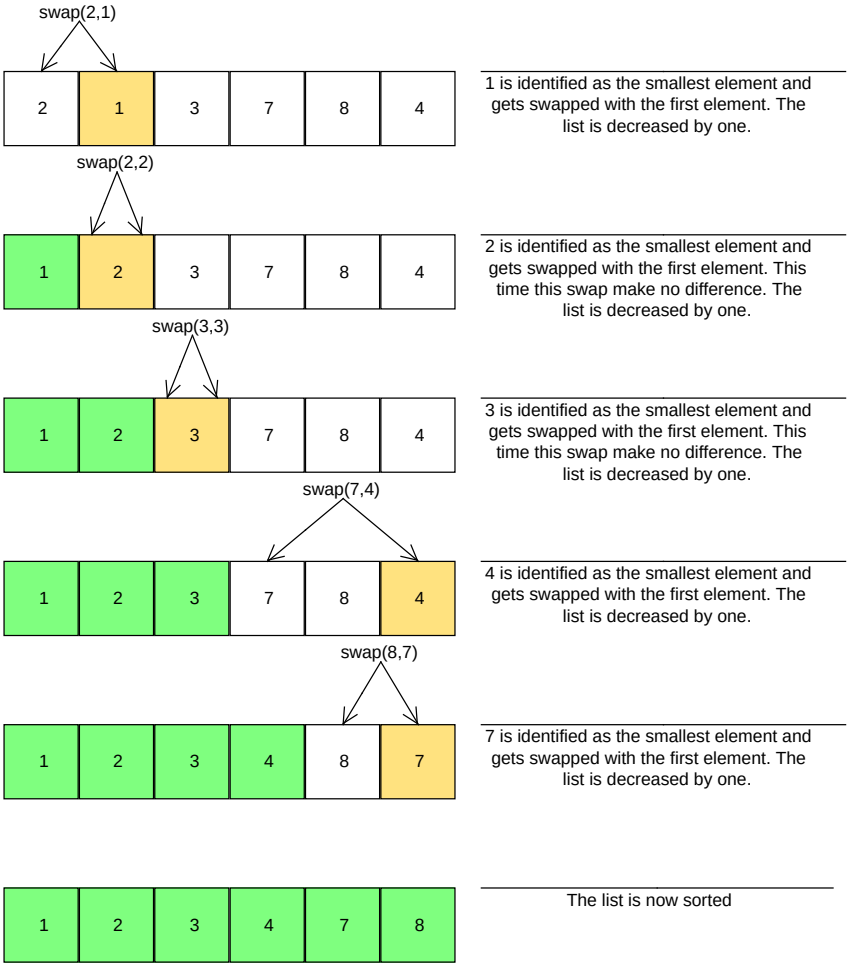

Figure 1: Selection sort step by step

2.4 Implementation

```
/**
 * \file SelectionSort.java
 * \brief
 * A simple implementation of the sorting algorithm
 * "selection sort".
 *
 * $Author: Mikael Tylmad$
 * $Created: 2008-02-15$
 */

public class SelectionSort
{

  /**
   * This method swaps two elements in a array of Objects.
   * \param theList The array which contains the elements to be swapped
   * \param a The first element
   * \param b The second element
   */
  private static void swap(Object[] theList,
                           int a,
                           int b)
  {
    Object tempElement = theList[a];
    theList[a] = theList[b];
    theList[b] = tempElement;
  }

  /**
   * This method sorts an array of comparable objects using the
   * "selection sort" algorithm.
   * \param theList The list which is to be sorted
   */
  public static void sort(Comparable[] theList)
  {
    int minIndex;
    int length = theList.length;

    // Loop over all the elements, except the last one. In each
    // loop, find the smallest value in the list (starting from
    // the index a) and swap this value with the one at the index a.
    for(int a=0;a<(length-1);a++)
    {
      minIndex = a;

      // find the index of the smallest value, starting from index a
      for(int b=a+1;b<length;b++)
        if(theList[minIndex].compareTo(theList[b]) > 0)
          minIndex = b;

      // make the swap
      if(minIndex != a)
        swap(theList,a,minIndex);
    }
  }

  // A simple test
  public static void main(String[] args)
  {
    SelectionSort.sort(args);

    for(String s : args)
      System.out.println(s);
  }
}
```

3 Bubble sort

3.1 The algorithm

Bubble sort is another simple algorithm. It works by repeatedly iterating through a list, comparing two elements at a time and swapping them if necessary. If an iteration through the list makes no swaps, the sorting is complete.

3.2 Complexity analysis

In the best case scenario where the list is already sorted, nothing is swapped and the running time is simply $\Theta(n)$. In the worst case however the list could be sorted backwards, which means that the algorithm would have to perform n swaps per iteration for n iterations. This equals a complexity of n^2.

There is a chance of improvement. After each iteration, the largest element will always be the n^{th} element. Therefore this element can be ignored at the next pass. This will shrink the list by one after each pass and quicken the algorithm. Instead of the worst case n^2 comparisons, the algorithm will now have a total of $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2$ comparisons. However this is in the same complexity scope, $\Theta(n^2)$.

3.3 Graphical example

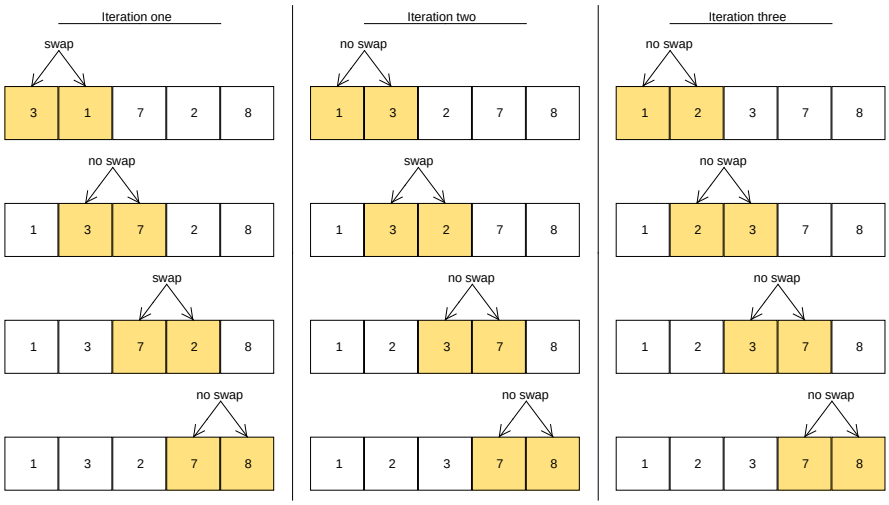

Figure 2: Bubble sort step by step

3.4 Implementation

```
/**
 * \file BubbleSort.java
 * \brief
 * A simple implementation of the sorting algorithm
 * "bubble sort".
 *
 * $Author: Mikael Tylmad$
 * $Created: 2008-02-18$
 */

public class BubbleSort
{

  /**
   * This method swaps two elements in a array of Objects.
   * \param theList The array which contains the elements to be swapped
   * \param a The first element
   * \param b The second element
   */
  private static void swap(Object[] theList,
                           int a,
                           int b)
  {
    Object tempElement = theList[a];
    theList[a] = theList[b];
    theList[b] = tempElement;
  }

  /**
   * This method sorts an array of comparable objects using the
   * "bubble sort" algorithm.
   * \param theList The list which is to be sorted
   */
  public static void sort(Comparable[] theList)
  {
    int lastIndex = theList.length - 1;
    boolean shouldMakeAnotherPass = true;

    // loop over the list repeatedly until no swap is made
    while(shouldMakeAnotherPass)
    {
      shouldMakeAnotherPass = false;

      for(int i=0;i<lastIndex;i++)
        if(theList[i].compareTo(theList[i+1]) > 0)
        {
          swap(theList,i,i+1);
          shouldMakeAnotherPass = true;
        }

      // The last element is always correctly sorted after each pass, so
      // the last index is decreased.
      lastIndex--;
    }
  }

  // A simple test
  public static void main(String[] args)
  {
    BubbleSort.sort(args);

    for(String s : args)
      System.out.println(s);
  }
}
```

4 Quicksort

4.1 The algorithm

Quicksort is a bit more complex than the previous algorithms. By employing a so called "divide and conquer" strategy, quicksort is able to sort a list with the low cost of $\Theta nlog(n)$. The algorithm works by following these steps:

1. Pick an element from the list. This element is called the *pivot value*
2. Reorder the list so that all elements smaller than the *pivot value* are positioned before the *pivot value*, and all the larger elements after. This operation is called *the partitioning*.
3. Regard the two newly created partitions, the inner lists before and after the *pivot value*, as separate lists. Recursively sort these with quicksort, which means beginning at step one again.

When selecting the *pivot value* in step one, it is important to remember that the optimal performance is achieved when selecting a value as close to the mean as possible. It is of course not possible to search after the perfect *pivot value* since this would severely damage the algorithms speed. A simple solution is to select the *pivot value* at random. Another is to choose a mean out of three elements in the list or even perhaps randomly mix the entire list before sorting it, making it possible to always select the first value as the *pivot value*.

4.2 Complexity analysis

When analyzing quicksort, the most simple approach is to begin with the partitioning process. This looks at all the elements in the list once and uses $\Theta(n)$ resources. Best case scenario is when the list is partitioned into two equally sized parts. Each recursive call will then operate on lists with the size $n/2$ which gives a call tree with a depth of $log(n)$. Now each level of the call tree will perform the partitioning process, but each of these calculations will only use $\Theta(n)$ resources. Since n is a lot smaller than in the beginning, these will all fit into the first $\Theta(n)$. This gives a resulting complexity of $\Theta(n \log(n))$.

In the worst case scenario, each partitioning will result in two lists with the sizes 1 and $n - 1$. In this case the call tree will no longer have a depth of $\log(n)$ but n. This will give a resulting complexity of $\Theta(n^2)$. However, when implementing quicksort with a randomized *pivot value* selection, the expected complexity is $\Theta(n \log(n))$.

4.3 Implementation

```
/**
 * \file Quicksort.java
 * \brief
 * A simple implementation of the sorting algorithm
 * "quicksort".
 *
 * $Author: Mikael Tylmad$
 * $Created: 2008-02-18$
 */

public class Quicksort
{

  /**
   * This method swaps two elements in a array of Objects.
   * \param theList The array which contains the elements to be swapped
   * \param a The first element
   * \param b The second element
   */
  private static void swap(Object[] theList,
                           int a,
                           int b)
  {
    Object tempElement = theList[a];
    theList[a] = theList[b];
    theList[b] = tempElement;
  }

  /**
   * This method partitions a selection in an array so
   * that all elements smaller than the pivot value are
   * moved to the left of the pivot value. All elements
   * larger are moved to the right.
   * \param theList The array which is to be partitioned
   * \param a The first element
   * \param b The second element
   */
  private static int partition(Comparable[] theList,
                               int leftIndex,
                               int rightIndex,
                               int pivotIndex)
  {

    // Remember the value of the pivot element and move
    // it out of danger
    Object pivotElement = theList[pivotIndex];
    swap(theList,pivotIndex,rightIndex);

    // Now, move all the elements smaller than the pivot
    // element to the front, remember where to insert the
    // next one, and later on: the pivot element.
    int newPivotIndex = leftIndex;
    for(int i=leftIndex;i<rightIndex;i++)
      if(theList[i].compareTo(pivotElement) < 0)
      {
        swap(theList,newPivotIndex,i);
        newPivotIndex++;
      }

    // Moving completed, lets put the pivot element where
    // it belongs and return the new index.
    swap(theList,rightIndex,newPivotIndex);

    return newPivotIndex;
  }

  /**
   * The recursive part of the quicksort algorithm. This method
   * first partitions the list so that a pivot value is located
   * at the correct position. It then calls itself with new indexes
   * that represent the two newly created partitions.
   * \param theList The list that is gradually sorted
```

```
   * \param leftIndex The current left index of the partition
   * \param rightIndex The current right index of the partition
   */
  private static void recursiveSort(Comparable[] theList,
                                    int leftIndex,
                                    int rightIndex)
  {
    // Ok, first check: is the recursion over?
    if(rightIndex > leftIndex)
    {
      // The recursion is not over, partition the list, the pivot value
      // is selected as the left value. This could be improved. After
      // partitioning, make two recursive calls, each sorting the
      // newly created partitions.
      int newPivotIndex =
        partition(theList,leftIndex, rightIndex,leftIndex);
      recursiveSort(theList,leftIndex,newPivotIndex-1);
      recursiveSort(theList,newPivotIndex+1,rightIndex);
    }
  }

  /**
   * This method sorts an array of comparable objects using the
   * "quicksort" algorithm.
   * \param theList The list which is to be sorted
   */
  public static void sort(Comparable[] theList)
  {
    recursiveSort(theList,0,theList.length-1);
  }

  // A simple test
  public static void main(String[] args)
  {
    Quicksort.sort(args);

    for(String s : args)
      System.out.println(s);
  }
}
```