Name: Jariya Phongsai

Class: MAC 286. Data Structure

Research Paper on Sorting Algorithms

Prof. Lawrence Muller

Date: October 26, 2009
**Introduction**

In computer science, a sorting algorithm is an efficient algorithm which perform an important task that puts elements of a list in a certain order or arrange a collection of items into a particular order. Sorting data has been developed to arrange the array values in various ways for a database. For instance, sorting will order an array of numbers from lowest to highest or from highest to lowest, or arrange an array of strings into alphabetical order. Typically, it sorts an array into increasing or decreasing order. Most simple sorting algorithms involve two steps which are compare two items and swap two items or copy one item. It continues executing over and over until the data is sorted.

Because sorting is important to optimizing the use of other algorithms in computer science such as binary search or linear search, it has been the subject of extensive research in computer science, and some very complexity methods have been developed.

**Sorting algorithms used in computer science are often classified by:**

- **System complexity of computational.** In this case, each method of sorting algorithm has different cases of performance, they are worst case, when the integers are not in order and they have to be swapped at least once. The term best case is used to describe the way an algorithm behaves under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. Average case is similar to worst case, but in average case, the integers are not in order.
- Computational complexity in terms of number of swaps. Sorting methods perform various numbers of swaps in order to sort a data.
- Memory usage is also a factor in classify the sorting algorithms.

Stability of sorting algorithms is used to classify the sorting algorithms. Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary.

Sorting algorithms are sometimes characterized by big O notation in terms of the performances that the algorithms yield and the amount of time that the algorithms take, where n is integer. Big O notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. Big O notation allows its users to simplify functions in order to concentrate on their growth rates. The different cases that are popular in sorting algorithms are:

- O(n) is fair, the graph is increasing in the smooth path.
- O(n^2): this is inefficient because if we input the larger data the graph shows the significant increase. It means that the larger the data the longer it will take.
- O(n log n): this is considered as efficient, because it shows the slower pace increase in the graph as we increase the size of array or data.
<table>
<thead>
<tr>
<th>Name</th>
<th>Average Case</th>
<th>Worst Case</th>
<th>Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>yes</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>O(n log n)</td>
<td>O(n^2)</td>
<td>no</td>
</tr>
<tr>
<td>Cocktail Sort</td>
<td>-</td>
<td>O(n^2)</td>
<td>yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>no</td>
</tr>
</tbody>
</table>
A few different algorithms that are popular in computer science are:

**Bubble Sort**

It is a straightforward and simple method sorting data that is used in computer science. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.

**Insertion sort**

It is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.

**Selection sort**

It is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.
Shell sort

Shell sort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort. Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements.

Quick sort

Quick sort is a divide and conquer algorithm which relies on a partition operation: to partition an array, we choose an element, called a pivot, move all smaller elements before the pivot, and move all greater elements after it. This can be done efficiently in linear time and in-place. We then recursively sort the lesser and greater sub lists. Efficient implementations of quick sort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, this makes quick sort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quick sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower O(n²) performance, but if at each step we choose the median as the pivot then it works in O(n log n).

Merge sort
It is a comparison-based sorting algorithm. In most implementations it is stable, meaning that it preserves the input order of equal elements in the sorted output.

**Cocktail sort**

Also known as bidirectional bubble sort, cocktail shaker sort, shaker sort (which can also refer to a variant of selection sort), ripple sort, shuttle sort or happy hour sort, is a variation of bubble sort that is both a stable sorting algorithm and a comparison sort. The algorithm differs from bubble sort in that sorts in both directions each pass through the list. This sorting algorithm is only marginally more difficult than bubble sort to implement, and solves the problem with so-called turtles in bubble sort.
**Objective**

In computer science, each sorting algorithm is better in some situation and has its own advantages. For example the insertion sort is preferable to the quick sort for small files and for almost-sorted files. To measure the performance of each sorting algorithm, the most important factor is runtime that a specific sort uses to execute a data. Because the fastest algorithm is the best algorithm, it pays to know which is the sorting algorithm fastest.

In this study, we will determine the efficiency of the various sorting algorithms according to the time and number of swaps by using randomized trails. The build environment will be built using the Java language. The research will discuss and implement several sorting algorithms such as bubble sort, selection sort, insertion sort and will also include complexity sort such as quick sort, cocktail sort, and merge sort. I will represents these algorithm as a way to sort an array or integers and run random trails of length. The research will provide the number of swaps and the runtime of each sorting algorithm.

To investigate, I create a package called “sorting” which contains two classes. First is called “sortingAlgorithms” which contains all sorting algorithms that can be called from any other classes. Another is “sortingTest” which is the class that we will use to test these sorting algorithms. The “sortingTest” class, will provide the amount of swaps of each sorting algorithm and the runtime (in millisecond) to execute a sort. In the experiment we will measure the runtime in millisecond because it can show the different
of each algorithm better than using second. Each algorithm takes short time to execute a sort. It takes less than a second for a big size of array (n=1,000,000).

Each of sorting algorithms works as follows:

**Bubble Sort**
- Exchange two adjacent elements if they are out of order. Repeat until array is sorted.

**Selection Sort**
- Find the smallest element in the array, and put it in the proper place. Swap it with the value in the first position. Repeat until array is sorted. (starting at the second position and advancing each time)

**Insertion Sort**
- Scan successive elements for an out-of-order item, then insert the item in the proper place.

**Quick Sort**
- Partition the array into two segments. In the first segment, all elements are less than or equal to the pivot value. In the second segment, all elements are greater than or equal to the pivot value. Finally, sort the two segments recursively.
Merge Sort

- Start from two sorted runs of length 1, merge into a single run of twice the length. Repeat until a single sorted run is left. Merge sort needs N/2 extra buffer.

Performance is second place on average, with quite good speed on nearly sorted array.

Shell Sort

- Sort every Nth element in an array using insertion sort. Repeat using smaller N values, until N = 1. On average, Shell sort is fourth place in speed. Shell sort may sort some distributions slowly.
Table: Comparison of the number of swaps relate to several sorting algorithms

<table>
<thead>
<tr>
<th>Sort</th>
<th>n=100</th>
<th>n=1000</th>
<th>n=10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>7890</td>
<td>879604</td>
<td>98530146</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>3146</td>
<td>243578</td>
<td>23453685</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>4913</td>
<td>267548</td>
<td>35648754</td>
</tr>
<tr>
<td>Cocktail Sort</td>
<td>4987</td>
<td>574970</td>
<td>45367598</td>
</tr>
</tbody>
</table>

**Note:** n= the number of integers in an array.
Table: Shows the average runtime of several sorting algorithms in different sizes of array.

<table>
<thead>
<tr>
<th>Sort</th>
<th>n=100</th>
<th>n=1000</th>
<th>n=10000</th>
<th>n=100000</th>
<th>n=1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>6</td>
<td>49</td>
<td>184</td>
<td>1749</td>
<td>2805</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>8</td>
<td>26</td>
<td>163</td>
<td>1239</td>
<td>15760</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>9</td>
<td>37</td>
<td>572</td>
<td>33853</td>
<td>35460</td>
</tr>
<tr>
<td>Cocktail Sort</td>
<td>7</td>
<td>57</td>
<td>174</td>
<td>2538</td>
<td>3740</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>2</td>
<td>13</td>
<td>43</td>
<td>918</td>
<td>1103</td>
</tr>
</tbody>
</table>

Table: Shows the worst runtime of several sorting algorithms in different sizes of array.

<table>
<thead>
<tr>
<th>Sort</th>
<th>n=100</th>
<th>n=1000</th>
<th>n=10000</th>
<th>n=100000</th>
<th>n=1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>8</td>
<td>73</td>
<td>697</td>
<td>54963</td>
<td>71402</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>11</td>
<td>42</td>
<td>227</td>
<td>21407</td>
<td>23647</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>12</td>
<td>63</td>
<td>940</td>
<td>34513</td>
<td>39214</td>
</tr>
<tr>
<td>Cocktail Sort</td>
<td>6</td>
<td>64</td>
<td>375</td>
<td>35488</td>
<td>42326</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>4</td>
<td>8</td>
<td>65</td>
<td>1984</td>
<td>2136</td>
</tr>
</tbody>
</table>
Diagram 1: The relationship between run time in average case and worst case for several sorting algorithms
(Integer (n) = 100)

Diagram 2: The relationship between run time in average case and worst case for several sorting algorithms
(Integer (n) = 1000)
Diagram 3: The relationship between run time in average case and worst case for several sorting algorithms
Integer (n) = 10000

Diagram 4: The relationship between run time in average case and worst case for several sorting algorithms
Integer (n) = 100,000
Diagram 5: The relationship between run time in average case and worst case for several sorting algorithms

Integer (n) = 1,000,000
**Conclusion and Speculation**

Base on our experiment, the charts displaying the relationship between execution time and array sizes show that the Quick sort is the fastest algorithm compared to Bubble sort, Insertion sort, Selection sort, and Cocktail sort. It also proves the literature about big O notation, that $O(n \log n)$ is considered efficient, because it shows the slower increase in the graph as we increase the size of the array. The diagram shows that Quick sort is efficient for both small and large integers. Although the worst case, it makes $O(n^2)$ comparison, typically, Quick sort is significantly faster in practice than other $O(n \log n)$ algorithms. On the other end of the efficiency scale, the Bubble sort is notoriously slow but is conceptually the simplest of the sorting algorithms and this is the reason a good introduction to sorting. In terms of swapping, the Bubble sort performs the greatest number of swaps because each element will only be compared to adjacent elements and exchanged if they are out of order. Cocktail sort is a slight variation of bubble sort. The table shows that the Cocktail sort performs fewer swaps than the Bubble sort. It is impossible because the Cocktail sort differs from the Bubble sort in that instead of repeatedly passing through the list from bottom to top, it passes alternately from bottom to top and then from top to bottom. It can achieve slightly better performance than a standard bubble sort. The reason for this is that bubble sort only passes through the list in one direction and therefore can only move items backward one step each iteration.

Insertion Sort sorts small array fast, but big array very slowly. Quick sort is fastest on average, but sometimes unbalanced partitions can lead to very slow sorting. Merge sort is stable in that two elements that are equally ranked in the array will not have their relative positions flipped.
References


package sorting;
import javax.swing.*;
public class sortingAlgorithms {
    //this class will contain sorting algorithms methods that can be called from any other class
    
    // this method adds the elements of an array of integers
    public static int addArray(int[] x) {
        int sum=0;
        for (int i=0; i<x.length; i++)
            sum += x[i]; // add the ith element to the sum
        return sum;
    }
    
    public static int findPosOfLargest(int[] x) {
        int pos=0; // assume largest is in the 0 position
        for (int i=1; i<x.length; i++)
            if (x[i] > x[pos])
                pos =i;
        return pos; // returns the position of the largest element
    }
    
    public static int findPosOfSmallest(int[] x) {
        int pos=0; // assume largest is in the 0 position
        for (int i=1; i<x.length; i++)
            if (x[i] < x[pos])
                pos =i;
        return pos; // returns the position of the smallest element
    }
    
    public static int findAverage(int[] x) {
        return addArray(x)/x.length;
    }
    
    public static void fillRandom(int[] x, int s, int e) {
        for (int i=0; i<x.length; i++)
            x[i]= (int) (Math.random() * (e-s +1) + s);
    }
    
    public static void fillArrayFromUser(int[] x) {
        for (int i=0; i<x.length; i++)
            // user input
    }
public static void printArray(int[] x, int[] s, int[] e) {
    for (int i = 0; i < x.length; i++)
        System.out.print(" "+x[i]);
    System.out.println();
}

private static void swap(int[] a, int sub1, int sub2) {
    // this method swaps the locations a[sub1] with a [sub2]
    int t = a[sub1];
    a[sub1] = a[sub2];
    a[sub2] = t;
}

public static void bubbleSort(int[] a) {
    // bubble sort with improvement
    int t = a.length - 2;
    int s = 0;
    long ctr = 0;
    for (int pass = 0; pass < a.length - 1; pass++)
    {
        if (s == t) break;
        else s = t;
        int i = -1;
        while (++i <= s)
            if (a[i] >= a[i + 1]){
                swap(a, i, i + 1);
                ctr++;
                // exchange value
                t = i;
            }
    }
    System.out.println("The data has swapped" + ctr);
}

public static void bubbleSort2(int[] a) {
    // simple bubble sort
    long ctr = 0;
    for (int pass = 0; pass < a.length - 1; pass++)
    {
        for (int i = 0; i < a.length - 1 - pass; i++)
            if (a[i] > a[i + 1]){
                swap(a, i, i + 1);
                ctr++;
            }
    }
    System.out.println("The data has swapped" + ctr);
}

public static void bubbleSort3(int[] a) {
    int floor = 0, // marker to the first unsorted
ceil = a.length-1; // marker to the last unsorted element
i, k;
long ctr=0;
Boolean swapMade = true;
while (swapMade) // keep looping as long as there's a swap being done
{
    swapMade = false;
    // assume there will be no swaps
    // forward bubble
    for (i = floor; i < ceil; i++)
    {
        if (a[i] > a[i+1])
        {
            swap (a, i, i+1);
            swapMade = true;
            ctr++;
        }
    }
    ceil = i;
}
System.out.println("count is" + ctr);

public static void selectionSort(int[] a)
{
    int t;
    long ctr = 0;
    for (int pass=0; pass<a.length-1; pass++)
    {
        int pos = sortingAlgorithms.findPosOfSmallest(a);
        // swap element of position with element of pass
        swap(a, pass, pos);
        ctr++;
    }
    System.out.println("The data has swapped" + ctr);
}

public static void cocktailSort(int[] a)
{
    int floor = 0;
    long ctr = 0;
    // marker to the first unsorted element
    int ceil = a.length-1;
    // marker to the last unsorted element
    int i, k;
    Boolean swapMade = true;
    while (swapMade) // keep looping as long as there's a swap being done
    {
        swapMade = false;
        // assume there will be no swaps
        // forward bubble
        for (i = floor; i < ceil; i++)
        {
            if (a[i] > a[i+1])
            {
                swap (a, i, i+1);
                swapMade = true;
            }
        }
    }
}
public static void insertionSort1(int[] a) {
    // O(n^2) algorithm very efficient for lists that are
    // almost sorted
    int j;
    long ctr = 0;
    for (int i=1; i<a.length; i++) {
        // find the proper location for the _ith_ element
        int t=a[i]; j=i-1;
        while (j>=0 && a[i]< a[j]) j--; // j is the position. we
        a[i] into
        for (int k=i-1; k > j; k--) // move over one position to
        the right
            a[k+1]=a[k];
        // move the _ith_ element or the _jth_ element
        a[j+1]=t;
        ctr++;
    }
    System.out.println("The data has swapped" + ctr);
}

public static void insertionSort2(int[] a, int st, int end) {
    // O(n^2) algorithm very efficient for lists that are
    // almost sorted
    int j;
    long ctr=0;
    for (int i=st +1; i<end+1; i++) {
        // find the proper location for the _ith_ element
        int t=a[i]; j=i-1;
        while (j>=0 && a[i]< a[j]) j--; // j is the position. we
        a[i] into
        for (int k=i-1; k > j; k--) // move over one position to
        the right
            a[k+1]=a[k];
        // move the _ith_ element or the _jth_ element
        a[j+1]=t;
    }
    System.out.println("The data has swapped" + ctr);
}
protected static int partition(int a[], int s, int e)
{
    medianOfThree(a, s, e);
    int ctr=0;
    int t=a[e]; // store pivot
    int lp=s-1;
    int rp=e;
    while (lp < rp)
    {
        do --rp; while (a[rp]>t); //stop rp if less or equal pivot
        do ++lp; while (a[lp]<t); //stop lp if greater or equal pivot
        swap(a,lp,rp);
        ctr++;
    }
    a[e]=a[rp]; // undo last swap
    a[rp]=a[lp]; // and move pivot
    a[lp]=t; // to final location
    return lp;
}

public static void shellSort(int a[])
{
int i, h=1;

// find the largest h value possible
do
    h = 3*h+1;
// while h remains larger than 0
while (h <= a.length);

do {
    // all sets h-sorted, now decrease set size
    h /= 3;
    // for each set of elements (there are h sets)
    for (i=h+1; i<= a.length; i++) {
        // pick the last element in the set
        int v = a[i];
        int j = i;
        // compare the element at V to the one before it in the set
        // if they are out of order continue this loop, moving elements "back" to make room for V to be inserted.
        while ((j>h) && (a[j-h] > v)) {
            a[j] = a[j-h];
            j -= h;
        }
        // insert V into the correct place
        a[j] = v;
    }
} while (h > 1);

// merge two arrays of integers keeping the order
public static void mergeIntArrays(int[] x, int[] y, int[] z){
    // combine all elements of x and y merging them into z
    int i=0, j=0, k=0; // i is subscript of x, j of y and k of z
    int ctr=0;
    for (;k<z.length; k++)
        if (i==x.length)
            // we finished the x array
            z[k]=y[j++];
        ctr++;

        if (j==y.length)
            z[k]=x[i++];
        else
            if (x[i] < y[j])
                z[k]=x[i++];
            else


public static int linearSearch(int[] a, int v) {
    // the search problem is to find the first occurrence
    // of v in array a
    // if no match return a.length
    // linear search is O(N)
    int i;
    for (i = 0; i < a.length; i++)
        if (a[i] == v)
            break;
    return i;
}

public static int linearSearch(int[] a, int v, int s) {
    // overloaded method
    // the search problem is to find the first occurrence
    // of v in array a
    // if no match return a.length
    // linear search is O(N)
    // begin search at position s
    // useful for finding all occurrences of v in a
    int i;
    for (i = s; i < a.length; i++)
        if (a[i] == v)
            break;
    return i;
}

public static int binarySearch(int a[], int s, int e, int v) {
    // find first occurrence of v in sorted array a
    // recursive implementation
    if (s > e)
        return a.length; // no match
    int g = (s + e) / 2; // always guess the middle element
    if (a[g] == v)
        return g; // we have a match
    else
        if (a[g] > v) // our guess is too big
            return binarySearch(a, s, g - 1, v); // throw out right
        else
            return binarySearch(a, g + 1, e, v); // throw out left
    }
**Testing class**

```java
package sorting;

public class sortingTest {

    public static void main (String [] args){
        long now= System.currentTimeMillis(); // to get the time at the beginning

        int []a= new int [100];
        // sortingData.fillRandom(a, 0, 100);
        sortingAlgorithms.fillRandom(a, 0, 100); // randomize the integers

        sortingAlgorithms.printArray(a, a, a); // To print the array
        sortingAlgorithms.bubbleSort(a); // can be changed to any sorting algorithms

        long end=System.currentTimeMillis(); // to get the ending time

        System.out.print("The sorting algorithm takes "+(end-now)+" millisecond"); // show the time that execute a database

    }
}
```