GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Benchmarking Sorting Algorithms

Student: Vladimir Novac

Date: 02.05.2020

Contents

The concept of sorting

Introduction

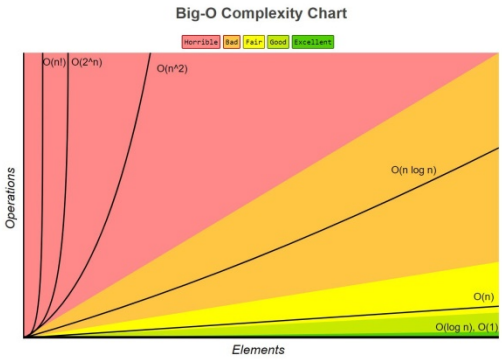
Sorting is one of the most fundamental algorithmic problem that was faced in the early days of computing. Most of the early computer science research was centered on finding a best way to sort a set of data. Supposedly, 25% of all CPU cycles are spent sorting, making sorting algorithms a very integral part of many software environments.
The process is basically comprised of taking a list of data, performing some permutations and arranging that data in a certain order. The most common orders are numerical or lexicographical.

Classification

Below is non exhaustive list by which sorting algorithms are classified:
Computational complexity - this describes the behaviour of the algorithm in terms of the size of the input list. Behaviour is noted as worst, average and best.
Memory usage – Algorithms that sort "in place" tend to use less memory that the ones which create a new structure.
Recursion – Some algorithms use recursive function or a non-recursive function or both.
Stability – A stable algorithm is considered to maintain the relative order of records with equal values.
Comparison Sort - Whether they are a comparison sort or not.
Method – the general method that is employed. Usual methods are: insertion, exchange, selection, merging.

Computational Complexity

We use the Big O notation to measure how long an algorithm takes to process an additional unit of input.
As the amount of resources required to run an algorithm generally varies with the size of the input, the complexity is typically expressed as a function.
The graph on the right shows the main notations for algorithmic performance in terms of time. The worst cases are highlighted in red, the average are in orange and the best cases are in yellow and green.



https://www.bigocheatsheet.com/

Examples of Sorting Algorithms

Some of the most popular sorting algorithms are:

- Quicksort
- Merge Sort
- Insertion Sort
- Counting Sort
- Selection Sort
- Bucket Sort
- Bubble Sort
- Tim Sort
- Comb Sort
- Heap Sort
- Radix Sort
- Cocktail Sort

In this project, I chose to test the following algorithms:
bubble sort, **insertion** sort, **selection** sort, **merge** sort and **counting** sort.

Bubble Sort

The bubble sort is one of the simplest algorithms used for sorting. It works by looping through all the elements in an array and swapping adjacent elements if they are in the wrong order.
The algorithm sorts the input array in place. Bubble sort is considered to be one of the worst sorting algorithms because of its low performance when dealing with anything higher than a small number of items to be sorted.

Type

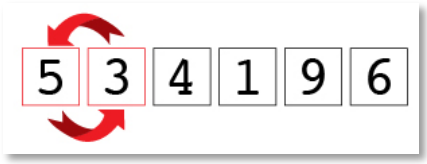
Comparison sort, sorting in place

Complexity

Worst Case - $O(n^2)$ Average Case - $O(n^2)$ Best Case - $O(n)$
Worst Case space complexity – $O(1)$

Implementation

Bubble sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.



Each pass, two adjacent elements are compared against each other. If the right one is bigger, the elements are swapped. If the right element is equal or smaller, it moves on to the next pair of elements.

The algorithm can be optimised by skipping over the last loop if no elements need swapping.

Java Implementation

```
public void bubbleSort(int[] inputArray) {
    boolean swapped;
    for (int i = 0; i < inputArray.length - 1; i++)
    {
        swapped = false;
        for (int j = 0; j < inputArray.length - i - 1; j++)
        {
            if (inputArray[j] > inputArray[j + 1])
            {
                // swap inputArray[j] and inputArray[j+1]
                int temp = inputArray[j];
                inputArray[j] = inputArray[j + 1];
                inputArray[j + 1] = temp;
                swapped = true;
            }
        }
        // IF no two elements were
        // swapped by inner loop, then break
        if (!swapped)
            break;
    }
}
```

Insertion Sort

The insertion sort is a sorting algorithm which builds a sorted array one element at a time. It has a simple implementation and is better suited for small sets of data. It is considered to be more efficient than bubble sort in practice and it is used for data sets that are already substantially sorted.

Type

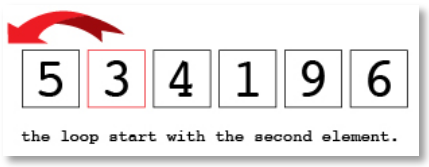
Comparison sort, sorting in place

Complexity

Worst Case - O(n^2) Average Case - O(n^2) Best Case - O(n)
Worst Case space complexity – O(1)

Implementation

Insertion sort iterates over each element in a list starting from the second one. On each iteration one element is removed from the initial list and added to the sorted list in its right place.



The loop starts with the second element.
On every iteration, the element is moved from its position and placed in the sorted array. If the element is larger than the previous one in the sorted list, it leaves the element in place and moves on to the next.
If the element is smaller, it takes the element from that position, searches through the sorted array and places the element in the right position by moving all the bigger values to the right.

After each iteration, the algorithm will jump to the next element in the list.

Java Implementation

```
    public void insertionSort(int[] inputArray) {
    int n = inputArray.length;
    for (int i = 1; i < n; ++i) {
        int key = inputArray[i];
        int j = i - 1;

        /* Move elements of inputArray, that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && inputArray[j] > key) {
            inputArray[j + 1] = inputArray[j];
            j = j - 1;
        }
        inputArray[j + 1] = key;
    }
}
```

Selection Sort

The selection sort is another simple comparison based algorithm that is deemed ineffective when working with large input data. It is considered to perform worse than the insertion sort algorithm under similar conditions.

Type

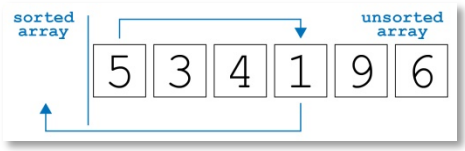
Comparison sort, sorting in place

Complexity

Worst Case - O(n^2) Average Case - O(n^2) Best Case - O(n^2)
Worst Case space complexity – O(1)

Implementation

Selection sort starts off by splitting the input list into a sorted and an unsorted list. Then it proceeds to check for the lowest number in the unsorted list and adds that number to the sorted list.



First, the array is split in two, the sorted and the unsorted array. The sorted array starts with zero elements.

Second, the algorithm searches for the lowest value in the unsorted array and moves that value at the beginning of the list and into the sorted array.

After the swap, the search index is incremented by one and selection will continue with the rest of the unsorted list until the list is exhausted.

Java Implementation

```
public void selectionSort(int[] inputArray)  {
    int n = inputArray.length;

    // One by one move boundary of unsorted sub array
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (inputArray[j] < inputArray[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = inputArray[min_idx];
        inputArray[min_idx] = inputArray[i];
        inputArray[i] = temp;
    }
}
```

Counting Sort

The counting sort algorithm sorts items according to keys stored as integers. Because of the key usage, this algorithm cannot be considered a comparison sort. It has a similar purpose as the bucket sort although the selection sort is much simpler to implement and uses less resources. Ideal usage include situations where the items in the list have similar values.

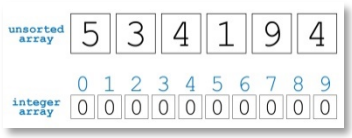
Type

Non-comparison sort, creates a new array

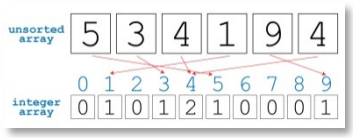
Complexity

Worst Case – O(n+k) Average Case - O(n+k) Best Case - O(n+k)
Worst Case space complexity – O(n+k) *where k is the range of the non-negative key values and n is the number of items*

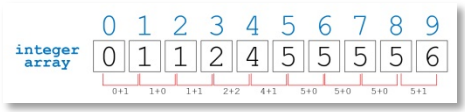
Implementation



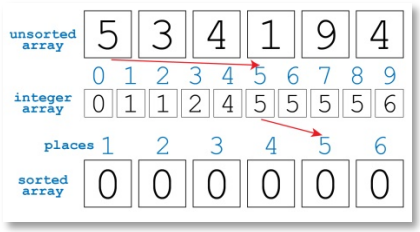
An integer array is created that will encompass the data range of the unsorted array



Each distinct value is counted and added in the integer array at the right place.



The values in the integer array are added together and a new sorted array is created that will have the length of the sum.



The algorithm loops through the unsorted array and for each element it finds the corresponding integer array index. The value from that index represent the place in the sorted array where the element will be inserted. After the placement, the value in the integer array will decrease by one.

Java Implementation

```
public void countSort(int[] inputArray) {
    int k = inputArray.length+1;
    // create an integer array of size n to store sorted array
    int[] output = new int[inputArray.length];
    // create an integer array of size k, initialized by all zero
    int[] freq = new int[k];
    // using value of integer in the input array as index,
    // store count of each integer in freq[] array
    for (int i: inputArray) {
        freq[i]++;
    }
    // calculate the starting index for each integer
    int total = 0;
    for (int i = 0; i < k; i++) {
        int oldCount = freq[i];
        freq[i] = total;
        total += oldCount;
    }
    // copy to output array, preserving order of inputs with equal keys
    for (int i: inputArray) {
        output[freq[i]] = i;
        freq[i]++;
    }
    // copy the output array back to the input array
    System.arraycopy(output, 0, inputArray, 0, inputArray.length);
}
```

Merge Sort

The merge sort algorithm is an efficient, general purpose algorithm that uses a divide and conquer strategy to sort items. The algorithm works by dividing the unsorted list into smaller lists until all lists have just one element. Then, these small lists are joined together to create a sorted list.

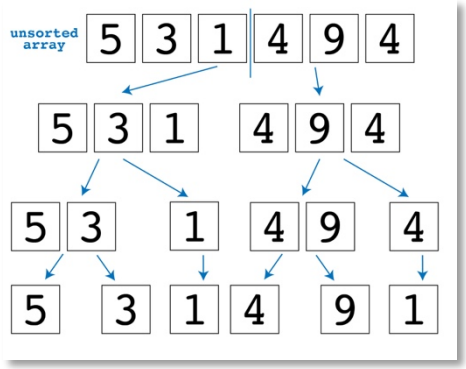
Type

comparison sort, divide and conquer, creates a new array

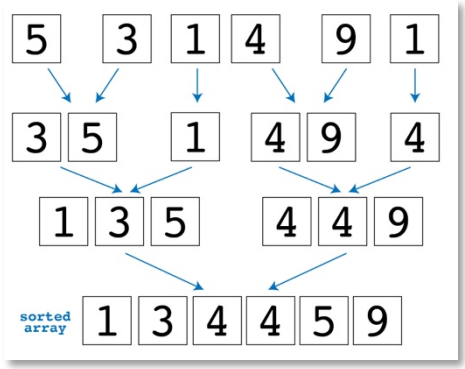
Complexity

Worst Case – O(n log n) Average Case - O(n log n) Best Case - O(n log n)
Worst Case space complexity – O(n)

Implementation



The middle point of the unsorted array is established and the array is split into two smaller arrays. The process is repeated until the arrays contain only one element.



The merging process begins by joining the one element arrays into larger sorted arrays. The process is repeated until the final sorted array is produced.

Java Implementation

```
public void merge(
        int[] a, int[] l, int[] r, int left, int right) {

    int i = 0, j = 0, k = 0;
    while (i < left && j < right) {
        if (l[i] <= r[j]) {
            a[k++] = l[i++];
        } else {
            a[k++] = r[j++];
        }
    }
    while (i < left) {
        a[k++] = l[i++];
    }
    while (j < right) {
        a[k++] = r[j++];
    }
}
```

```
public void countSort(int[] inputArray) {
    int k = inputArray.length+1;
    // create an integer array of size n to store sorted array
    int[] output = new int[inputArray.length];
    // create an integer array of size k, initialized by all zero
    int[] freq = new int[k];
    // using value of integer in the input array as index,
    // store count of each integer in freq[] array
    for (int i: inputArray) {
        freq[i]++;
    }
    // calculate the starting index for each integer
    int total = 0;
    for (int i = 0; i < k; i++) {
        int oldCount = freq[i];
        freq[i] = total;
        total += oldCount;
    }
    // copy to output array, preserving order of inputs with equal keys
    for (int i: inputArray) {
        output[freq[i]] = i;
        freq[i]++;
    }
    // copy the output array back to the input array
    System.arraycopy(output, 0, inputArray, 0, inputArray.length);
}
```

Implementation & Benchmarking

Java Implementation

In this project, I chose to test the following algorithms: **bubble** sort, **insertion** sort, **selection** sort, **merge** sort and **counting** sort. I set the number of iterations for each algorithm to 10 and the data set number to 13.

```
// number of iterations for each algorithm
int iterations = 10;
// number of random sets for each iteration
int setNumber = 13;
```

To hold the random data, I generated 5 independent array lists which I filled with random integers according to the data sets.

```
for (int i = 0; i < iterations; i++) {
    arrListBubble.add(generateRandomArray(new int[setNumber][]));
    arrListSelection.add(generateRandomArray(new int[setNumber][]));
    arrListInsert.add(generateRandomArray(new int[setNumber][]));
    arrListCounting.add(generateRandomArray(new int[setNumber][]));
    arrListMerge.add(generateRandomArray(new int[setNumber][]));
}
```

I created 5 arrays of arrays of type double to store the time for each algorithm. After this, I ran each algorithm with its own random data ten times. Each time the algorithm would run, a different set of random data would be executed.

```
for (int i = 0; i < iterations; i++) {
    timeElapsedBubble[i] = runAlgorithm(arrListBubble.get(i), 1);
    timeElapsedSelection[i] = runAlgorithm(arrListSelection.get(i), 2);
    timeElapsedInsert[i] = runAlgorithm(arrListInsert.get(i), 3);
    timeElapsedCounting[i] = runAlgorithm(arrListCounting.get(i), 4);
    timeElapsedMerge[i] = runAlgorithm(arrListMerge.get(i), 5);
}
```

When all the tests were finished, I averaged the times stored in each array to get the final times for each data set.

```
//get an average number for each algorithm from the 10 iterations
double[] bubble = averageArray(timeElapsedBubble);
double[] selection = averageArray(timeElapsedSelection);
double[] insert = averageArray(timeElapsedInsert);
double[] counting = averageArray(timeElapsedCounting);
double[] merge = averageArray(timeElapsedMerge);
```

The results were then printed to the console.

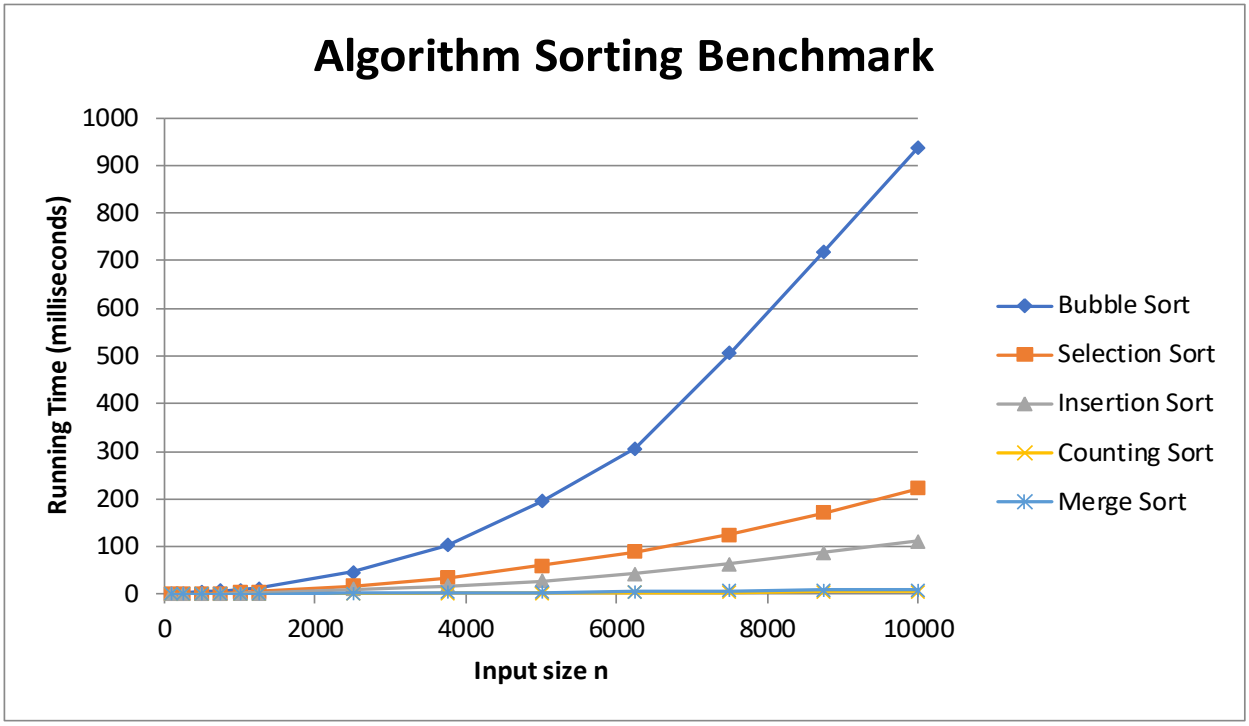
```
//print the results to the console
printResults(bubble, selection, insert, counting, merge);
```

```
---------- Sorting Algorithms Benchmark ----------
-          Generating Random Arrays...          -
-     Running Algorithms with test data...      -
-                    Done                       -
-------------------------------------------------


-------------------------------------------------------------------------------------------------------------------------
 Size           100     250     500     750     1000    1250    2500    3750    5000    6250    7500    8750    10000
 Bubble Sort    3.044   2.114   5.913   7.953   9.810   12.699  46.091  103.267 194.433 306.132 505.372 720.195 938.881
 Selection Sort 0.236   0.705   1.287   2.484   3.932   4.942   16.891  35.278  59.727  89.141  125.985 171.053 222.291
 Insertion Sort 0.092   0.280   0.701   1.328   2.175   2.939   11.071  18.178  27.610  42.827  64.213  87.376  111.704
 Counting Sort  0.060   0.135   0.261   0.392   0.521   0.668   1.330   2.051   2.759   3.494   4.177   4.917   5.640
 Merge Sort     0.244   0.303   0.577   0.744   0.899   1.137   2.312   3.552   4.740   5.897   7.388   8.405   9.380
-------------------------------------------------------------------------------------------------------------------------

Process finished with exit code 0
```

Benchmark Results

Sample results table – all values are in milliseconds, and are the average of 10 repeated runs.
The top row shows the number of random integers tested.

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	3.044	2.114	5.913	7.953	9.810	12.699	46.091	103.267	194.433	306.132	505.372	720.195	938.881
Selection Sort	0.236	0.705	1.287	2.484	3.932	4.942	16.861	35.278	59.727	89.141	125.985	171.053	222.291
Insertion Sort	0.092	0.280	0.701	1.328	2.175	2.939	11.071	18.178	27.610	42.827	64.213	87.376	111.704
Counting Sort	0.060	0.135	0.261	0.392	0.521	0.668	1.330	2.051	2.759	3.494	4.177	4.917	5.640
Merge Sort	0.244	0.303	0.577	0.744	0.899	1.137	2.312	3.552	4.740	5.897	7.388	8.405	9.380



The graph above shows the running times of the algorithms on average from 10 iterations.
The x-axis on the chart measures the input size of the list. The y-axis measures time in milliseconds.

Summary

All algorithms behave more or less the same when sorting a small number of input data. There is very little difference between them up until the 1500 mark. From then on, we can see a definite difference in their growth.
As shown in the graph, bubble sort grows at the fastest rate then the other algorithms tested.
Selection sort comes in second and insertion sort is third.
Lastly, the counting sort is obscured by merge sort but both show very similar results.
As it was to be expected, counting sort is the fastest of the five. Due to the small number of input integers (max 10000), merge sort comes in very close to counting sort.
This test has shown results that are in line with other, more comprehensive tests carried out.