Sorting Algorithms
and benchmarking their performance in Java
Computational Thinking with Algorithms, 2021

Executive Summary

An Java application was written to benchmark five different sorting algorithms: Bubble Sort, Quicksort, Counting Sort, Insertion Sort and Selection Sort. Their relative performance was evaluated and Bubble Sort was found to have the worst performance with regards to time complexity for all input sizes, whereas Counting Sort generally performed most favourably out of the group.

Glossary

- 'a' and 'b' will be used throughout the document for explanation purposes to describe elements in an array.
- k(Table 2) refers to the maximum possible value in the input array/ the range of input.
- n refers to the number of elements in the input array/the size of the array.

Introduction

Sorting algorithms are a set of rules that take in an array or list and arrange the elements into a particular order. This process is advantageous as computations can be made simpler if information is sorted in advance. (Heineman et al., 2008).

Sorting Algorithm Classification

Figure 1 below shows some of the common factors taken into account when classifying sorting algorithms.

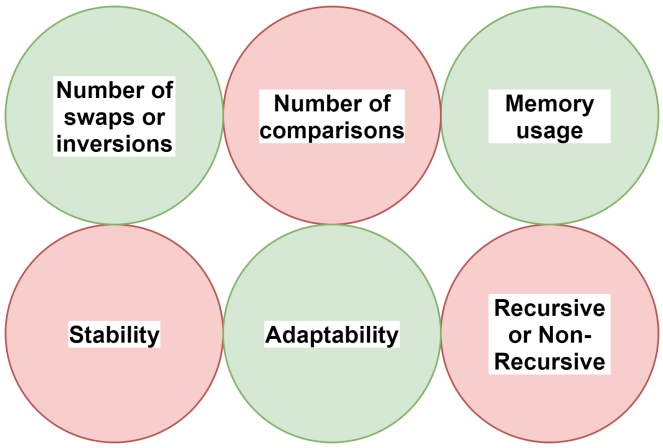

Figure 1. Sorting Algorithm Classification

- Number of swaps or inversions: This is the number of times an algorithm must swap elements to successfully sort data.
- Number of comparisons: This is the number of times an algorithm must compare elements in order to sort data. (Free Code Camp, 2020).
- Memory usage: The memory requirements of a sorting algorithm depends on how it works. In-place sorting algorithms require a fixed additional amount of memory regardless of the input size. Therefore if memory is limited an in-place algorithm is the best option. An out-of-place algorithm requires extra memory and the amount is based on input size. (Free Code Camp, 2020).
- Stability: An algorithm's stability is based on how it deals with elements of equal value. If input data is already sorted, the algorithm should preserve this order. (Heineman et al., 2008).
- Adaptability: An algorithm can be described as adaptable if it takes advantage of existing order within the input data to improve its running times. (Geeks For Geeks, 2021).
- Recursion: Algorithms can be recursive or non-recursive. There are also algorithms which use a combination of both. (Free Code Camp, 2020).

Comparator Functions

Sorting Algorithms require a method of ordering elements , this can be achieved by using comparator functions. The process is based on the following: for any two elements in a collection, one of the following is true: a=b, a<b, or a>b. The comparator function returns zero if a=b, a negative number if a<b, and a positive number if a>b. Input in the form of integers and floating point numbers, for example, can easily be compared in this way. (Heineman et al., 2008). Input in the form of strings of characters, for example, can be compared by applying a lexicographical ordering system to each individual character in the string. (Mannion, 2021 b).

Analysing sorting algorithms

There are several factors which must be taken into consideration when analysing and assessing an algorithm's complexity:

- The most expensive computation in terms of time or space will determine its overall classification.
- There may be multiple instances of size n and therefore if assessing two algorithms, the first may outperform the second in a particular instance of the problem, but the second may outperform the first in another instance of the same problem. There is no one-size-fits-all solution. (Mannion, 2021 b).
- The underlying computational problem being solved and the characteristics of the data to be sorted will impact complexity, therefore complexity should be evaluated in the best, average and worst cases. (w3schools, n.d.).

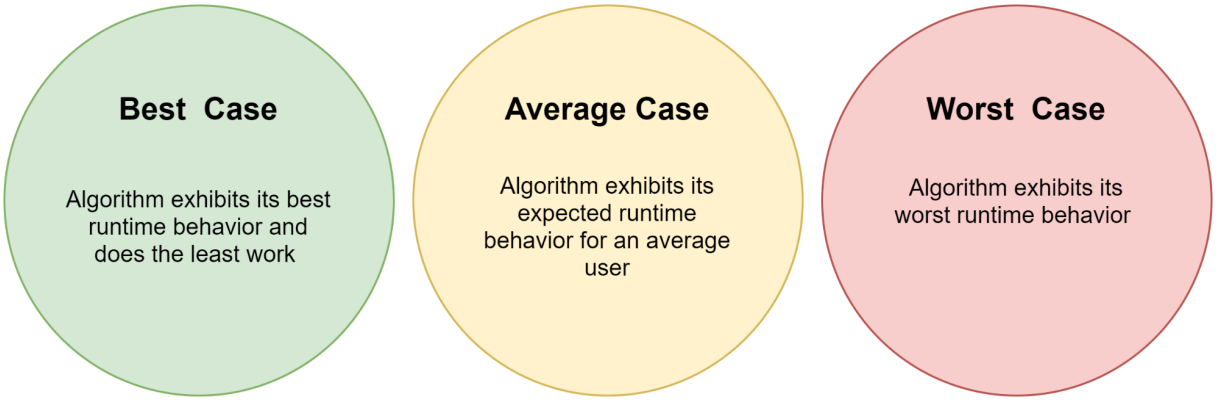

Figure 2. Algorithm Best, Average, and Worst Case descriptors

Considering an algorithm's performance in each of these cases allows the most appropriate algorithm to be selected for various situations.

Worst case is the maximum execution time taken over all instances of size n. Big O notation, in simple terms, is a measure of how fast a function grows or declines and is used to describe algorithm complexity in the worst case. The less complex an algorithm is with regards to Big O, the more efficient it is. (Towards Data Science, n.d.).

The average case is usually the most challenging to accurately measure and to do involves advanced mathematical techniques. It also assumes input may be partially sorted.

Ω(omega) notation is used to describe algorithm complexity in the best case. It should be noted this most likely will not occur often. It is a measure of the least number of potential operations. (Mannion, 2021 c).

Algorithm complexity can be classified by order of growth into the families in Figure 3 and Table 1 below: (adapted from lecture notes Mannion, 2021 b).

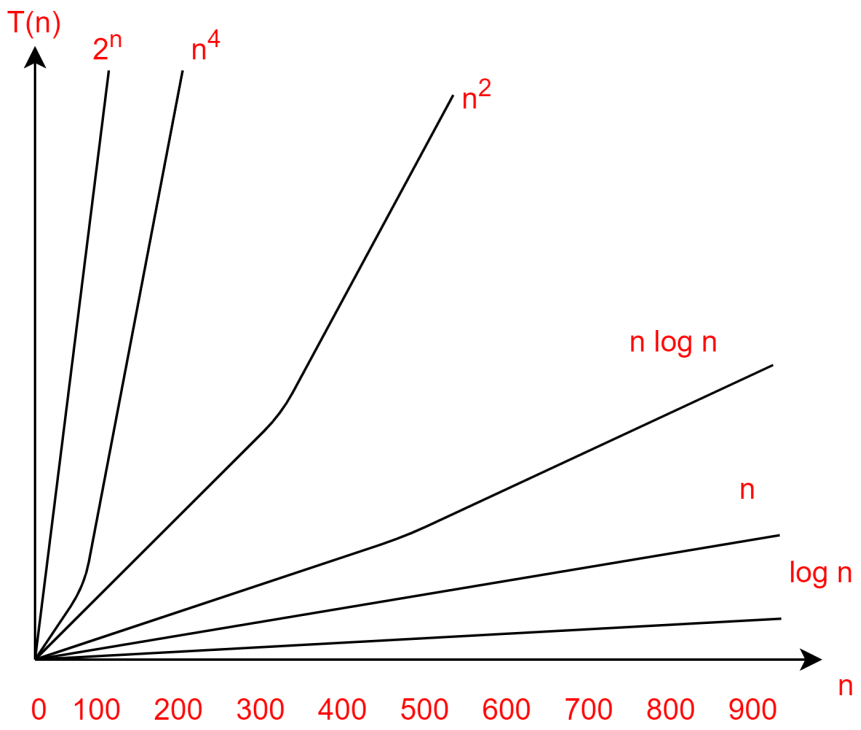

Figure 3. Typical Complexity Curves

Running time T(n) is proportional to:	Complexity Family
T(n) α log n	logarithmic
T(n) α n	linear
T(n) α n log n	linearithmic
T(n) α n^2	quadratic
T(n) α n^3	cubic
T(n) α n^k	polynomial
T(n) α 2^n	exponential
T(n) α $k^n; k > 1$	exponential

Table 1. Running time in proportion to complexity

Algorithms have different time and space efficiencies. Time efficiency focuses on the amount of time it takes for an algorithm to execute. This is important as an algorithm should accomplish its goal in an acceptable time frame. Space efficiency focuses on the amount of memory required for an algorithm to execute. (Mannion, P., 2021 a).

Comparison type sorting algorithms

Comparison sorts can be applied to a wide range of data input types. These sorts only use comparison operations to decide in which order two elements should occur in a sorted list. Elements can be checked against and sorted based on a property known as the sort key i.e. we use the sort key to decide on the ordering. (Heineman et al., 2008).

Simple comparison-based sorts include Bubble Sort, Selection Sort, and Insertion Sort. Efficient comparison-based sorts include Quicksort, Merge Sort and Heap Sort.

No algorithm that sorts by comparing elements can do better than n log n performance in the average or worst cases. (Mannion, P., 2021 b).

Non- comparison type sorting algorithms

Non-comparison sorts work by making assumptions about the input data, comparison sorts don't. An example of the type of assumption made is if the input data is within a certain range. This property allows for the possibility of an O(n) time complexity result.

Non-comparison sorts include Counting Sort and Bucket Sort.

Features of a 'good' sorting algorithm

There are a variety of different sorting algorithms, each with their own pluses and minuses. A desirable sorting algorithm should exhibit the following:

- Good run time efficiency (in best, average or worst case).
- In-place sorting (if memory concern).
- Suitability i.e. the properties of the sorting algorithm should match well with the class of input instances expected. (Mannion, 2021 c).

Sorting Algorithms included in the benchmarking process

The following five algorithms were selected for benchmarking in the application:

1. Bubble Sort
2. Quick Sort
3. Counting Sort
4. Insertion Sort
5. Selection Sort

Algorithm	Best Case	Worst Case	Average Case	Space Complexity	Stable
Bubble Sort	n	n^2	n^2	1	Yes
Quicksort	n log n	n^2	n log n	n (worst case)	No*
Counting Sort	n + k	n + k	n + k	n + k	Yes
Insertion Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No

* Standard Quicksort is unstable however there are stable versions of Quicksort available

Table 2. Overview of the Sorting Algorithms included in the benchmarking process (adapted from (Mannion, 2021 c).

Bubble Sort - a simple comparison based sort

Overview

Bubble Sort is a comparison-based, in-place, stable sorting algorithm. Its time complexity is n in best case, and n^2 in the worst and average cases. (Mannion, 2021 c).

How it works

- Bubble Sort works by comparing each element with the element on its right, excluding the last element. If the elements in the pair are out of order, it swaps them. This results in the largest element being put at the very end and into its correct place, hence it will not be moved again. (Zobenica, n.d.).
- In the Figure 4 below this procedure is followed and in step 1 element 11, the largest element, is moved to the far right/end of the collection and sorted into its final place
- Next each element (except the last two) are compared with the one to the right and if they are out of order, they are swapped.
- This puts the second largest element next to last and the last two elements are now in their correct and final places.
- This process continues until there are no remaining unsorted elements on the left.

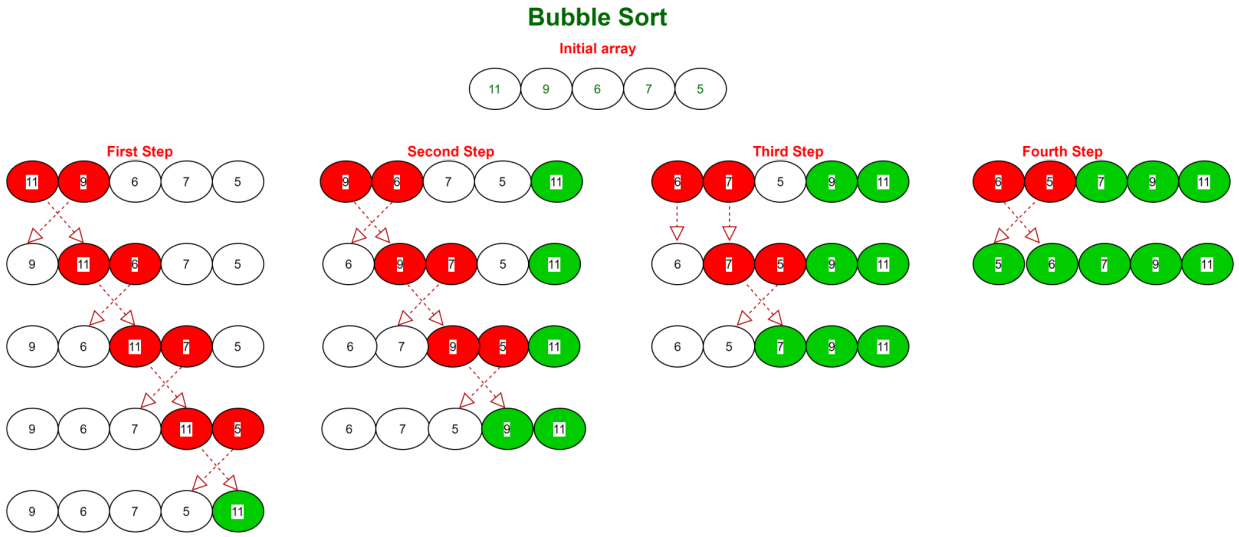

Figure 4. Bubble Sort example **(adapted from Mannion, 2021c)**

```
static void bubbleSort(int[] arr) {
    int n = arr.length;
    int temp = 0;

    for (int i = 0; i < n; i++) { //Outer loop
      for (int j = 1; j < (n - i); j++) { //Inner loop
        if (arr[j - 1] > arr[j]) {
          //swap
          temp = arr[j - 1];
          arr[j - 1] = arr[j];
          arr[j] = temp;
        }
      }
    }
}
```

Figure 5. Bubble Sort example in Java code (adapted from JavaTPoint, n.d.)

Advantages

Bubble Sort is simple to understand and implement and can be practical if data is nearly sorted.

Disadvantages

It is slow and impractical for most problems even when compared to Insertion Sort.

Quicksort - an efficient comparison based sort

Overview

Quicksort is a comparison based, in-place, unstable sorting algorithm. Its time complexity is n log n in the best case, n^2 in the worst case and n log n in the average case.

How it works

- Quicksort uses a divide and conquer method. An element is chosen to be the 'pivot' element and the array to be sorted is partitioned around this pivot. (Goodrich & Tamassia, 2005).
- The pivot element can be decided upon in a variety of ways depending on the version of Quicksort used e.g. always choosing the first element or last element as pivot, choosing the median element as pivot or choosing a random element as pivot. (Heineman et al., 2008).
- The next step, partitioning, involves changing the order of the elements so that elements smaller than the pivot are placed before it and elements greater than the pivot are placed after it. This results in the pivot being in its final position.
- Then the resultant two subarrays can then be recursively sorted.
- In Figure 6 below the first element of the initial array and the resultant subarrays is chosen each time.
- The recursion aspect of Quicksort requires a base case. The base case can be a subarray of either length 1 or 0 as a single element array requires no sorting. (Mannion, 2021 d).

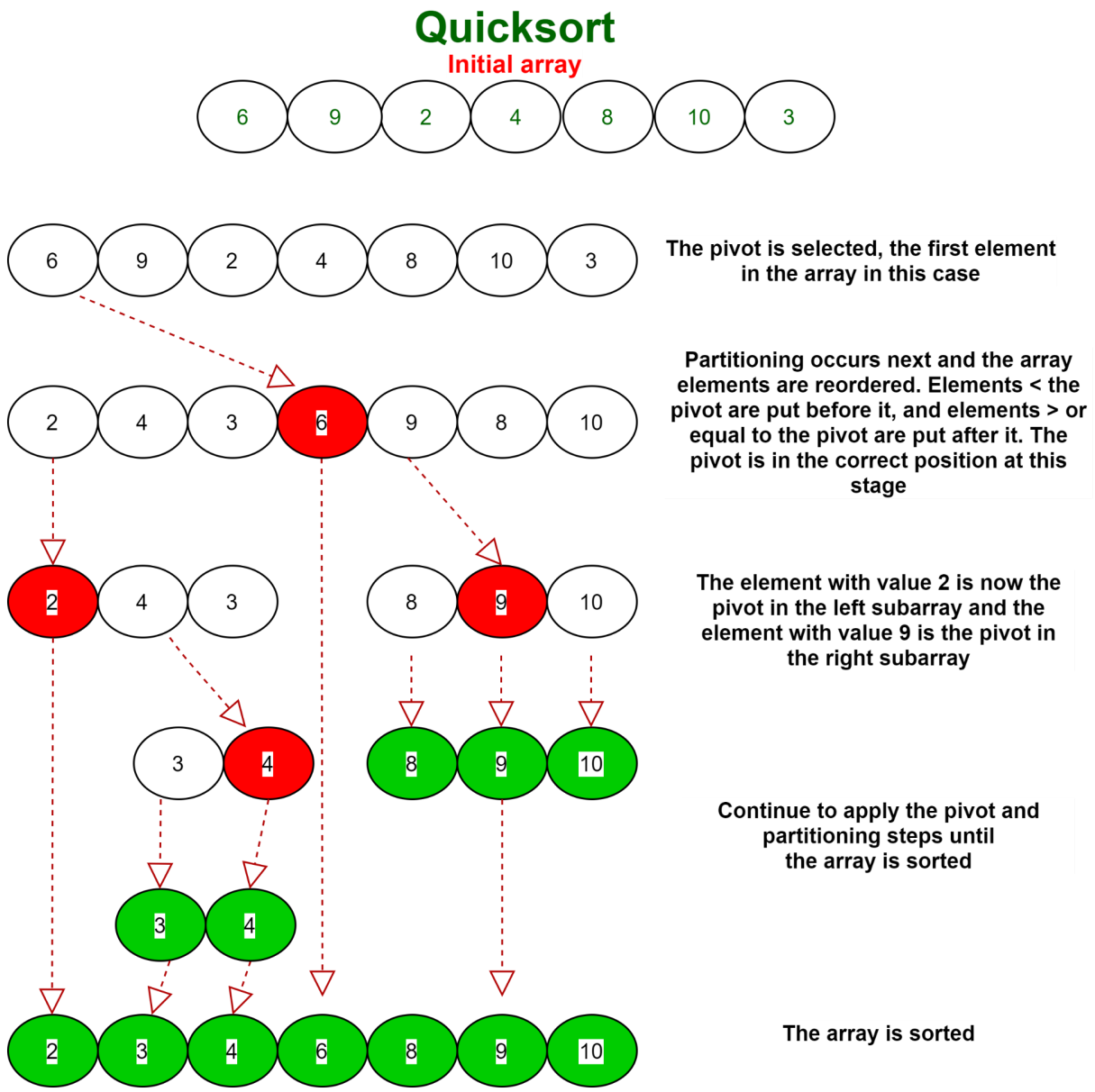

Figure 6. Quicksort example (adapted from Mannion, 2021d and HackerRank, n.d.)

```
public class QuickSort {

  private int temp_array[];
  private int lng;

  public void sort(int[] nums) {
    // This if statement provides the base case for recursion
    if (nums == null || nums.length == 0) {
      return;
    } //Otherwise proceed:
    this.temp_array = nums;
    lng = nums.length;
    quickSort(0, l - 1);
  }
   private void quickSort(int low_index, int high_index) {
    int i = low_index;
    int j = high_index;
    // The pivot is calculated here
    int pivot = temp_array[low_index+(high_index-low_index)/2];
    // And partitioning occurs
    while (i <= j) {
        while (temp_array[i] < pivot) {
         i++;
       }
       while (temp_array[j] > pivot) {
         j--;
       }
       if (i <= j) {
         exchangeNumbers(i, j);
         i++;
         j--;
       }
    }
    /*
      The quickSort method is called here recursively i.e. it calls
      itself. The pivot and partitioning steps are applied recursively
      to the subarrays
    */
    if (low_index < j)
      quickSort(low_index, j);
    if (i < high_index)
      quickSort(i, high_index);
  }

  private void exchangeNumbers(int i, int j) {
    int temp = temp_array[i];
    temp_array[i] = temp_array[j];
    temp_array[j] = temp;
  }
}
```

Figure 7. Quicksort example in Java code (adapted from w3schools, 2020)

Advantages
Quicksort generally is one of the fastest sorting algorithms as noted in the research performed by Verma & Singh in 2015.

Disadvantages
It is unstable.

Counting Sort - a non comparison sort

Overview
Counting Sort is a non-comparison, out-of-place, stable sorting algorithm with n + k time complexity in the best, worst and average cases.

How it works

- Counting Sort works on the principle that certain assumptions are made about the input data i.e. it assumes that the data will fall within a range.
- The first step involves calculating the key range k of the input array.
- A key step in sorting the data is determining how many times every unique key value occurs in the input array and so an array is initialised to store this information, the count array in Figure 8 below.
- A location to store the output is required and so an output array of size n is then initialised.
- The algorithm then iterates through the input array.
- The output array is based on the unique key frequencies identified. (Mannion, 2021 d, Heineman et al., 2008).

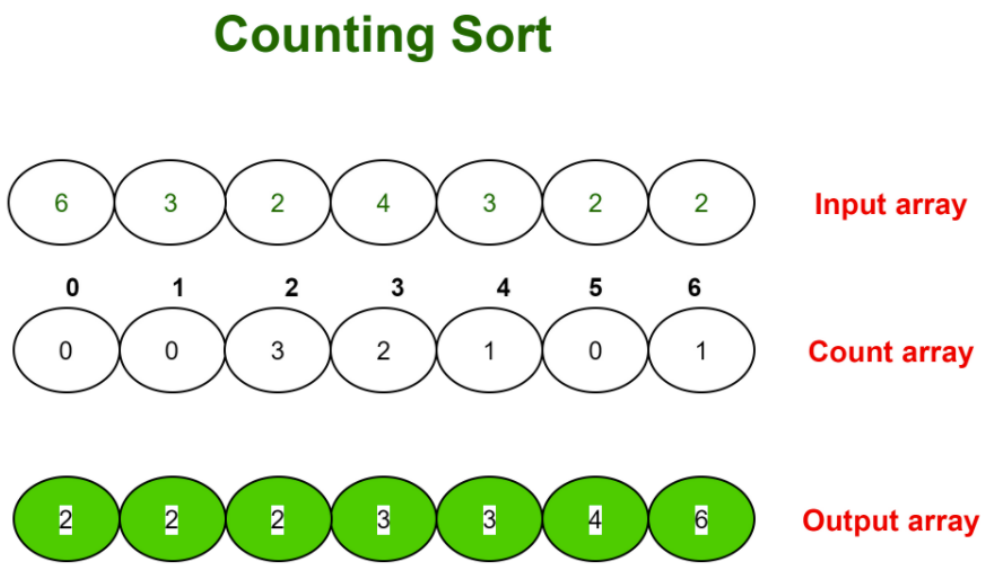

Figure 8. Counting Sort example (adapted from Mannion, 2021 d)

```
void countSort(int array[], int size) {
  int[] output = new int[size + 1];

 //Use a for loop to find the max element and so determine the key range k
  int max = array[0];
  for (int i = 1; i < size; i++) {
    if (array[i] > max)
      max = array[i];
  }
  int[] count = new int[max + 1]; //Create the count array
  for (int i = 0; i < max; ++i) {
    count[i] = 0;
  }
  // Store the count of every unique key value
  for (int i = 0; i < size; i++) {
    count[array[i]]++;
  }
  /*
    Iterate through the count array to get the result and store it in the
    output array
  */
  for (int i = size - 1; i >= 0; i--) {
    output[count[array[i]] - 1] = array[i];
    count[array[i]]--;
  }
}
```

Figure 9. Counting Sort example in Java Code (adapted from Programiz, n.d.)

Advantages
Counting Sort allows for sorting in close to linear time.

Disadvantages
It is only useful in limited situations as it is only suitable for input instances of a fixed range. (Mannion, 2021 d).

Insertion Sort - a simple comparison based sort

Overview
Insertion Sort is a comparison based, in-place, stable sorting algorithm with n in the best and n^2 in the worst and average cases of time complexity.

How it works

- Insertion Sort divides a list of size n into two sublists: the head which is sorted and the tail which is unsorted.
- The element at index 1 is set as the key. Any elements to the left of the key which are greater in value than the key must be moved to the right by one position. Then insert the key. In Figure 10 below index 1 contains the value 9 and so this is set as the key, 11 is to the left of the key and larger than it and so moves one position to the right. The key is then inserted into index 0.
- For the next step follow the same procedure however this time the element at index 2 is set as the key and again any elements to the left of the key which are greater than the key must be moved to the right by one position. Then insert the key.
  This process is continued until the element at index n-1 is set as the key. Any elements to the left of the key which are greater in value than the key must be moved to the right by one position. Then insert the key. The array is sorted after this step. (Mannion, 2021 c), (Heineman et al., 2008).

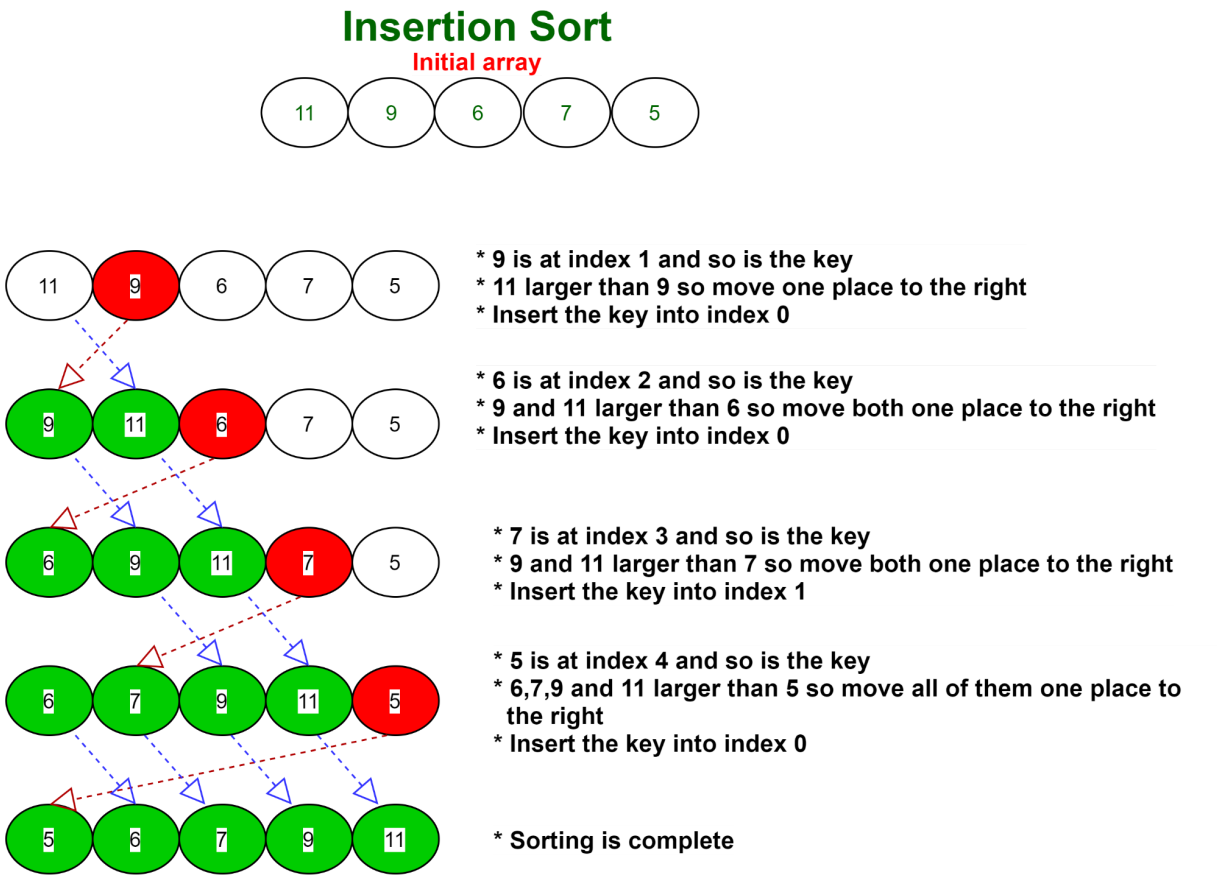

Figure 10. Insertion Sort example

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
      int key = arr[i]; //Declare the key
      int j = i - 1;

      /*
        While loop to find elements to the
        left of the key which are > in value
        than it
      */
      while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j = j - 1;
      }
      arr[j + 1] = key; //Put key into new position
    }
  }
```

Figure 11. Insertion Sort example in Java code (adapted from lecture notes Mannion, 2021 c)

Advantages
Insertion Sort is easy to implement and suitable for small lists or lists that are in an almost sorted state

Disadvantages
It is inefficient for larger data input sizes.

Selection Sort - a simple comparison based sort

Overview
Selection Sort is a comparison based, in-place, unstable sorting algorithm with n^2 complexity in the best, worst and average cases.

How it works

- Selection Sort works using two subarrays: an unsorted on the right and a sorted subarray on the left. For each iteration the minimum element (if using ascending order) is moved from the unsorted to the sorted subarray.
- In Figure 12 below the first step involves iterating through the elements at index 0 through to n-1 and selecting the smallest. This smallest element is then swapped with the element in index 0.
- The second step involves the same process except this time the elements at index 1 through to n-1 are searched. The smallest is again selected and it is swapped with the element in index 1.
- This process is continued until there are no elements left to search. (Heineman et al., 2008, Tutorials Point, n.d.).

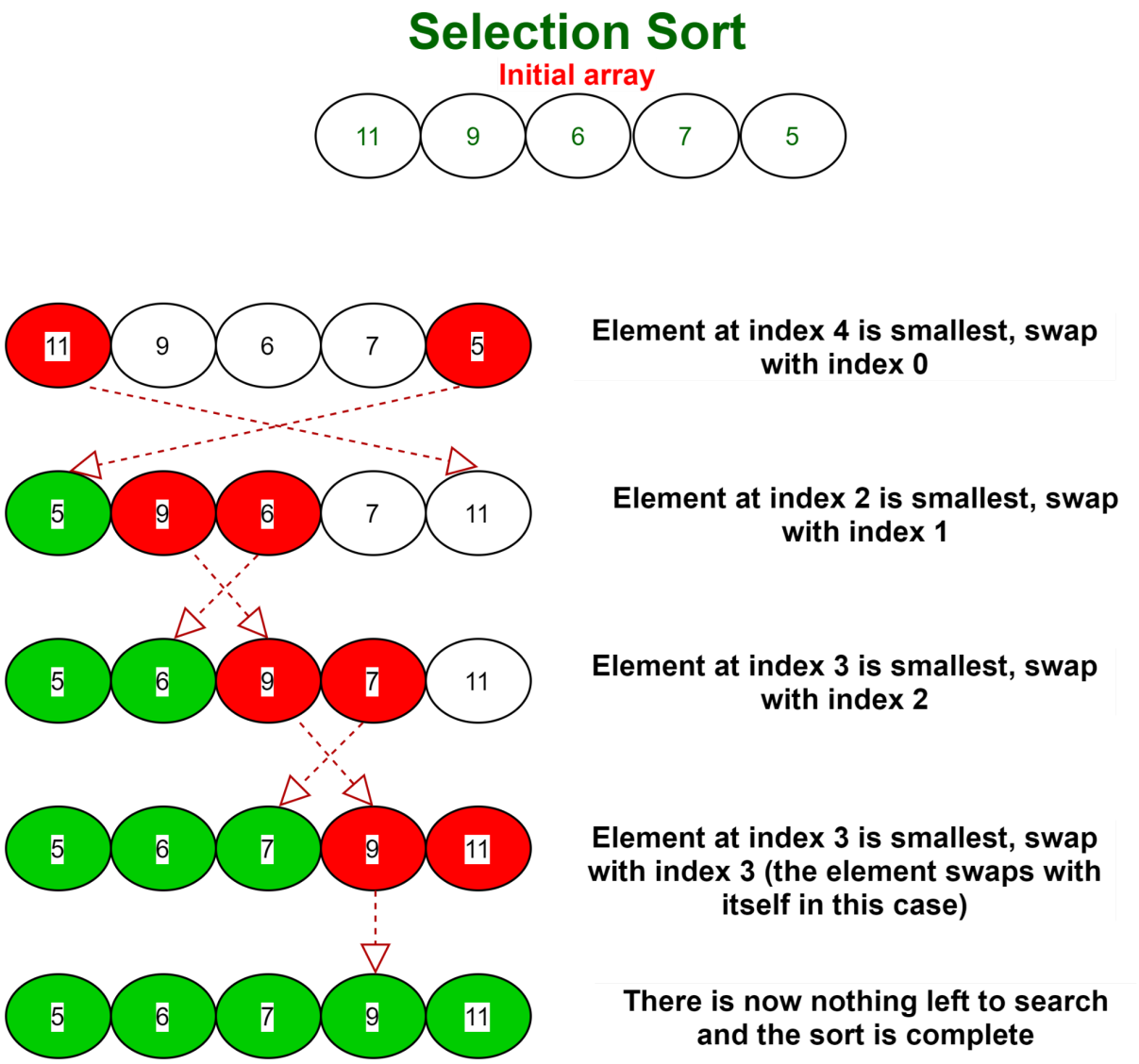

Figure 12. Selection Sort example (adapted from lecture notes Mannion, 2021 c)

```java
public static void selectionSort(int[] arr) {

    int outer = 0, inner = 0, min = 0;

    //Outer loop increments
    for (outer = 0; outer < arr.length - 1; outer++) {
       min = outer;
       for (inner = outer + 1; inner < arr.length; inner++) {
          //If statement to find the min value index
          if (arr[inner] < arr[min]) {
             min = inner;
          }
       }
       //Swap the min value with the outer value
       int temp = arr[outer];
       arr[outer] = arr[min];
       arr[min] = temp;
       //Continue until no elements left to search
    }
}
```

Figure 13. Selection Sort example in Java code (adapted from lecture notes Mannion, 2021 c)

Advantages

Selection Sort does not require a large amount of memory for sorting as it is in-place

Disadvantages

It is unsuitable for significant input data sizes.

Implementation and Benchmarking

Benchmarking is a posteriori analysis which uses empirical methods to compare the performance of algorithms, sorting algorithms in this case. (Mannion, 2021 d).
Experimental data has a part to play in proving that theoretical data is well founded. The data must be calculated in a precise way so that the results of analysis are accurate. (Heineman et al., 2008). One way in which this can be achieved is by running multiple tests under the same conditions to ensure consistency.
Algorithms are platform independent i.e. can run on a variety of different architectures and so the results of empirical testing are dependent on the platform they are implemented on.
A platform independent method of algorithm complexity comparison can be achieved by assessing the scalability of algorithms and so scalability was used as a measure in this benchmarking process.
Scalability can be determined by assessing the complexity of the algorithm with regards to an input data of size n. It should be noted that memory needs can also be assessed in this way. (Mannion, 2021 d).

The benchmarking tests were run on a Dell XPS 13 9310 P.C. with 16GB of RAM and an 11th Generation Intel(R) Core i7-1165G7 processor using Windows OS Build 19042.928. The application was written using JDK 16 and implemented in the IntelliJ IDEA integrated development environment.

The application included implementations of the five chosen sorting algorithms along with a main method to test each one. It also included a 'TestHarness' class to manage data gathering and output of results and a 'Timer' class to record the running times for each algorithm.

Arrays of randomly generated integers with different input sizes n were used to benchmark the algorithms. The running time for each algorithm was measured 10 times and the average of these 10 measurements was output to the console for analysis.

Analysis/Discussion

The results of the benchmarking analysis were similar to expected when taking known theoretical data into account. In Figure 14 below it is evident that Bubble Sort is much slower in terms of time complexity than other sorting algorithms. Table 3 shows that Counting Sort consistently outperformed the other algorithms and this was to be expected in this case as the input data falls within a certain range.

Input Size	100	1000	2500	5000	10000	20000	50000	100000
CountingSort	0	0	0	0	1	0	2	1
InsertionSort	0	0	1	3	7	20	205	457
QuickSort	0	7	4	18	57	211	2361	5686
SelectionSort	0	9	20	68	166	666	7668	18882
BubbleSort	1	12	33	183	949	3665	36491	160130

Table 3. Results of the benchmarking analysis in table form

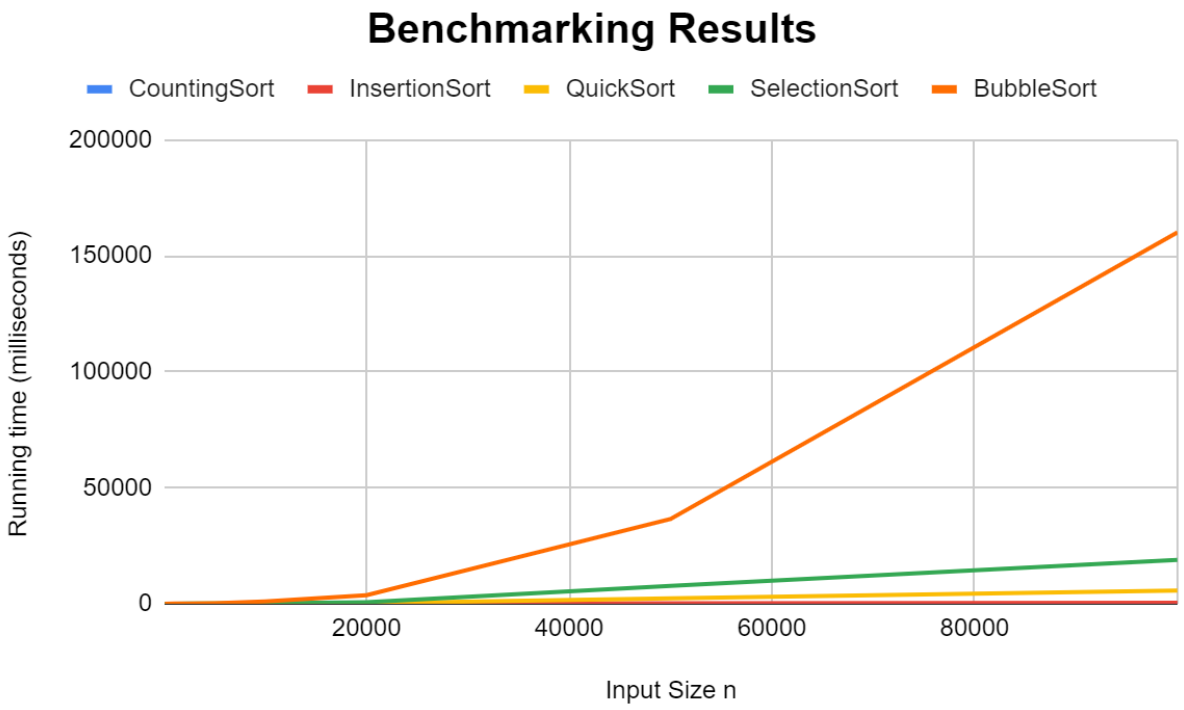

Figure 14. Results of the benchmarking analysis in graph form

Selection Sort performed better than Bubble Sort however was much slower than the other algorithms and this poor performance was expected as it exhibits only O(n^2) even in the best case.
On small arrays Insertion Sort is faster than Quicksort (Heineman et al., 2008) and that is reflected in the results of this analysis. Insertion Sort exhibits O(n + m) behaviour where m is the number of inversions and therefore is most effective for small input data sizes, for situations where this is not the case it exhibits undesirable O(n^2) behaviour. (Goodrich & Tamassia, 2005).
Quicksort is poor for real time applications. (Goodrich & Tamassia, 2005). It does exhibit O(n log n) in the best case however its complexity depends on the pivot selection, if partitioning in each recursive step results in one empty subarray and one large subarray the algorithm will exhibit worst-case quadratic behaviour. In the ideal case, partition will result in the original array being divided in half. (Heineman et al., 2008).
Alternatives not benchmarked in this analysis include Merge Sort, this is an out-of-place algorithm and is not suitable where memory is limited but if an external memory device can be used this is a desirable option. (Goodrich & Tamassia, 2005).

Conclusions

When choosing a sorting algorithm the following facts should be taken into consideration:

- Bubble Sort is very slow and therefore impractical.
- Selection Sort usually outperforms Bubble Sort.
- Insertion Sort is unsuitable for large, random lists but if the input size is very small or if the data is sorted or nearly sorted, it can outperform algorithms with a greater complexity.
- Quicksort is good in the average case.
- Counting Sort outperformed the other algorithms in this analysis however it is known to be useful in input instances of a fixed range and this was the type of input data set used here.

Even if an algorithm has a preferable best-case, average-case or worst-case time complexity, its use may prove impractical. There is no single sorting algorithm that fits the bill for all data input instances. Each algorithm has its own associated benefits and shortcomings and selection of an appropriate algorithm must be made on a case by case basis.

References

Free Code Camp (2020, January 18). *Sorting Algorithms Explained.* Sorting Algorithms Explained (freecodecamp.org)

Geeks For Geeks (2021, March 17). *Classification of Sorting Algorithms.* https://www.geeksforgeeks.org/classification-of-sorting-algorithms/

Goodrich, M.T., & Tamassia, R. (2005). *Data Structures and Algorithms in Java* (4th ed). John Wiley & Sons Inc

HackerRank (n.d.). *Quicksort 2 - Sorting.* https://www.hackerrank.com/challenges/quicksort2/problem

Heineman, G., Pollice, G., & Selkow, S. (2008). *Algorithms in a nutshell* (1st ed). O'Reilly Media.

Java T Point (n.d.). *Bubble Sort in Java*. https://www.javatpoint.com/bubble-sort-in-java

Mannion, P. (2021)a. Analysing Algorithms, Part 2. [Lecture notes pdf]. Retrieved fromAnalysing Algorithms Part 2 (gmit.ie)

Mannion, P. (2021)b. Sorting Algorithms, Part 1. [Lecture notes pdf]. Retrieved from Sorting Algorithms Part 1 (gmit.ie)

Mannion, P. (2021)c. Sorting Algorithms, Part 2. [Lecture notes pdf]. Retrieved from Sorting Algorithms Part 2 (gmit.ie)

Mannion, P. (2021)d. Benchmarking Algorithms in Java. [Lecture notes pdf]. Retrieved from11 Benchmarking in Java (gmit.ie)

Programiz (n.d.). *Counting Sort Algorithm.* https://www.programiz.com/dsa/counting-sort#:~:text=Counting%20sort%20is%20a%20sorting,index%20of%20the%20auxiliary%20array.

Towards Data Science (n.d.). *Introduction To Big O Notation.* https://towardsdatascience.com/introduction-to-big-o-notation-820d2e25d3fd

Tutorials Point (n.d.). *Data Structure and Algorithms Selection Sort.* https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

Verma, R., & Singh, J. (2015). A comparative analysis of deterministic sorting algorithms based on runtime and count of various operations. *International Journal of Advanced Computer Research,* 5(21), 380-385.

w3schools(2020). *Java Exercises Quicksort Algorithm.* https://www.w3resource.com/java-exercises/sorting/java-sorting-algorithm-exercise-1.php

w3schools (n.d.). *Sorting Techniques.* https://www.w3schools.in/data-structures-tutorial/sorting-techniques/

Zobenica, D.(n.d.). *Sorting Algorithms in Java.* https://stackabuse.com/sorting-algorithms-in-java/