Sorting in an array

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the fourth assignment "Sorting an array" on the course "ID2021" TIDAB, KTH. The assignment is to explore three different sorting algorithms and their effectiveness, this will be evaluated inform of an evaluation of run-time as a function of the size of an array that we will be sorting. All of the arrays will have randomized elements form 0 to 1000 and will be evaluated many times to get a stable value of the measurement. The array is randomized for for each time a sorting function is called to counter behind the scenes optimization from the IDE or Java. The creation of this array is not included in the time benchmark.

Selection sort

Selection sort is a sorting algorithm that starts with first value in an array and then goes through the whole array to find the smallest element smaller than the index value to swap the position of these elements. The function then repeats this for every position in the array. If no smaller element is found it simply moves to the next position in the array. The algorithm is very easy to implement but not so efficient on larger arrays. The first element needs to be compared n-1 times, the second n-2 times and the rest follows this pattern which gives the equation $(n-1)+(n-2)+...+1 = n(n-1)/2$ for the number of comparisons. This gives the *Selection sort* algorithm a time complexity of $O(n^2)$. Table 1 shows the measurements for *Selection sort*, the code for the algorithm is shown below:

```
for (int i = 0; i < array.length -1; i++){
     int cand = array[i];
     int temp;
     for (int j = i; j < array.length; j++){
         if (array[j] < cand){
             temp = array[j];
             array[j] = cand;
```

```
                cand = temp;
            }
        array[i]= cand;
        }
    }
```

Repetitions(n)	**10**	**100**	**1000**	**10000**	**100000**
Run-time(ns)	$3 * 10^4$	$2 * 10^6$	$7,6 * 10^8$	$4,2 * 10^11$	$1,4 * 10^14$

Table 1: Mesurements of the run-time as a function of array size for *Selection sort*. Run-time in nano seconds.

Insertion sort

Insertion sort is an algorithm where one element is taken per repetition from the start of the array and then inserted in the where it belongs in comparison to all previous elements that have been sorted. Starting from left to right the unsorted list gradually turns into a sorted list. The best case scenario is when the input array is already sorted which gives the algorithm a linear time complexity of $O(n)$. The worst case is when the array is sorted backwards to every element has to be compared and moved to the left. This results in the time complexity of $O(n^2)$. These findings can give insertion sort advantages when an array is short or partly sorted from the beginning. Table 2 shows the measurements for *Insertion sort*, code for the sorting algorithm is given below:

```
for(int i = 1; i < array.length; i++){
            int cand = array[i];
            int j = i - 1;
            while( j>=0 && array[j] > cand){
                array[j+1] = array[j];
                j--;
            }
            array[j+1] = cand;
        }
```

Repetitions(n)	**10**	**100**	**1000**	**10000**	**100000**
Run-time(ns)	$5 * 10^3$	$1 * 10^4$	$5,2 * 10^5$	$5,2 * 10^7$	$5,3 * 10^9$

Table 2: Mesurements of the run-time as a function of array size for *Insertion sort*. Run-time in nano seconds.

Merge sort

Merge sort is a sorting algorithm where the array is divided into two parts and then these are combined in a sorted manner. The algorithm is based on the *Divide and Conquer* paradigm where you split a problem into many sub problems to solve and then combine them into a solution. The algorithm is recursive and continuously splits the array in half until all elements are separated. Then begins the process of merging where the halves are added into a new array, but in a sorted order. Since the array is continuously divided the length of the binary tree that is formed will be $log(n)$ and the number of operations per branch is equal to n so the time complexity of merge sort is $n * log(n)$. Table 3 shows the measurements for *Merge sort*, the code for the sorting algorithm is shown below:

```
private static void mergeSort(int[] array){
    if (array.length < 2 )
    return;
    int mid = array.length/2;
    int[] left = new int[mid];
    int[] right = new int[array.length - mid];
    for (int index = 0; index < mid; index++){
        left[index] = array[index];
    }
    for (int index = mid; index < array.length; index++){
        right[index - mid] = array[index];
    }

    mergeSort(left);
    mergeSort(right);
    merge(array, left, right);
}

private static void merge(int[] array, int[] left, int[] right){
    int leftLen = left.length;
    int rightLen = right.length;
    int indexLeft = 0, indexRight = 0, indexMerge = 0;
    while (indexLeft < leftLen && indexRight < rightLen){
        if (left[indexLeft] <= right[indexRight]){
            array[indexMerge] = left[indexLeft];
            indexLeft++;
        }
        else{
            array[indexMerge] = right[indexRight];
            indexRight++;
```

```
            }
            indexMerge++;
        }
        while (indexLeft < leftLen){
            array[indexMerge] = left[indexLeft];
            indexLeft++;
            indexMerge++;
        }
        while (indexRight < rightLen){
            array[indexMerge] = right[indexRight];
            indexRight++;
            indexMerge++;
        }
    }
```

Repetitions(n)	**10**	**100**	**1000**	**10000**	**100000**
Run-time(ns)	$5,2 * 10^3$	$1,3 * 10^4$	$1,4 * 10^5$	$1,3 * 10^6$	$1,5 * 10^7$

Table 3: Mesurements of the run-time as a function of array size for *Merge sort*. Run-time in nano seconds.

Conclusion

In conclusion sorting an array takes a lot of run-time in comparison to what we have done previously in the course. The first two algorithms *Selection sort* and *Insertion sort* were easy to implement while *Merge sort* was harder to implement and required a lot more code. From the bench-marking results *Merge sort* was the winner of the three overall, but interestingly when the array was shorter *Insertion sort* was the fastest. This can be explained by increased complexity of *Merge sort* which has several conditional statements and the lower variation in values where the time complexity of *Insertion sort* on average can come closer to $O(n)$. When the tests were done with lower variety of random numbers an improvement could also be seen for *Insertion sort*. Merge sort seems to be a great sorting algorithm for arrays that is still stable with a very large array size whereas the other two could be useful for smaller scale arrays or when memory usage could be affected negatively by the *Merge sort* algorithm