Sorting an array

Axel Månson Lokrantz

Spring Fall 2022

Introduction

In the previous assignment we worked with arrays looking for elements, this time we shift focus to sorting the actual arrays through different algorithms. The program was written in Java in collaboration with a few fellow students.

Task 1: Selection Sort

The first algorithm is called selection sort, an in-place comparison algorithm that is easy to setup but not very efficient for larger lists. The algorithm sorts an array by finding the smallest element from the unsorted part and putting it in the beginning of the array.

```
    public static int[] sort (int[] array){

        for (int i = 0; i < array.length -1; i++){
            int index = i;

            for(int j = i; j < array.length; j++){
                if (array[j] < array[index])
                index = j;
            }
            int temp = array[index];
            array[index] = array[i];
            array[i] = temp;
        }
        return array;
    }
}
```

To decide what element should be placed on the first position $n-1$ element comparisons are done, for the second position $n-2$ and so forth. The number of comparisons decrease by one for each position which gives the following formula $(n-1)+(n-2)+...+1 = n(n-1)/2$. $n*n$ is dominant while the rest is considered negligible which results in a performance of $O(n^2)$.

n	ns
128	7597
256	22091
512	71197
1024	281065
2048	1080590
4096	3958784
8192	15406489
16384	60683390

Table 1: Selection sort benchmark.

Task 2: Insertion Sort

Insertion sort is a simple sorting algorithm that repeatedly searches items to find the smallest item and places it in the correct position. The main difference between the two algorithms is that insertion sort builds the final sorted list by transferring one element at a time.

```java
public static int[] sort(int[] array){
    int temp;
    int holePos;
    for (int i = 1; i < array.length; i++){
        holePos = i;
        temp = array[i];
        while(holePos > 0 && temp < array[holePos -1]){
            array[holePos] = array[holePos -1];
            holePos--;
        }
        array[holePos] = temp;
    }
    return array;
}
```

Best case occurs when the array is already sorted, or close to. In total that is n-1 comparisons which gives a linear performance of $O(n)$. Worst case occurs when the array is sorted in reverse, meaning every element except the first one is compared to all elements in the array. $1+2+3+...+(n-2)(n-1)$ which results in the performance of $O(n^2)$. So which of the two algorithms should be used and when? Insertion sort perform much better on arrays that are mostly sorted at the start. While selection sort always performs $O(n)$, insertion sort has an average and worst case with the performance of $O(n^2)$. Therefor selection sort is preferable if writing to memory is more expensive than reading.

n	ns
128	2383
256	6717
512	22812
1024	75774
2048	277423
4096	1056497
8192	4140513
16384	16799803

Table 2: Insertion sort benchmark.

Task 3: Merge sort

The last sorting algorithm is called merge sort, which is a recursive algorithm. A little more complex than the previous ones merge sort continuously splits the array into two halves. When there is only one element left the algorithm repeatedly merges elements into new sorted sub-arrays to gradually sort the entire array.

```
public static void sort(int[] array){
        int inputLength = array.length;
        if (inputLength < 2){
            return;
        }
        int midIndex = inputLength / 2;
        int[] leftHalf = new int[midIndex];
        int[] rightHalf = new int[inputLength - midIndex];

        for (int i = 0; i < midIndex; i++){
            leftHalf[i] = array[i];
        }
        for (int i = midIndex; i < inputLength; i++){
            rightHalf[i - midIndex] = array[i];
        }
        sort(leftHalf);
        sort(rightHalf);
        merge(array, leftHalf, rightHalf);
```

```
    }
    private static void merge (int[] array, int[] leftHalf, int[] rightHalf){
        int leftSize = leftHalf.length;
        int rightSize = rightHalf.length;
        int i = 0, j = 0, k = 0;
        while (i < leftSize && j < rightSize){
            if (leftHalf[i] <= rightHalf[j]){
                array[k] = leftHalf[i];
                i++;
            }
            else{
                array[k] = rightHalf[j];
                j++;
            }
            k++;
        }
        while (i < leftSize){
            array[k] = leftHalf[i];
            i++;
            k++;
        }
        while (j < rightSize){
            array[k] = rightHalf[j];
            j++;
            k++;
        }
    }
}
```

n	ns
128	7818
256	16345
512	30957
1024	63463
2048	141142
4096	289655
8192	566044
16384	1083167

Table 3: Merge sort benchmark.

As seen in table 1,2 and 3 merge sort is the clear winner out of the three algorithms. Insertion sort is about three times faster than selection sort while merge sort is about 20% faster than insertion sort when running benchmarks on an array of 1024 elements. Merge sort is a recursive algorithm which can be described with the following formula: $2(n/2) + O(n)$ where 2(n/2) corresponds to the time it takes to sort all the sub-arrays and O(n) the time it takes to merge the entire array. Simplified the time complexity becomes $O(n(log(n))$ which is the same for the worst, average and best cases since the algorithm always divides and merges the array.