Quick sort

Axel Månson Lokrantz

Spring Fall 2022

Introduction

This far we have worked with several sorting algorithms such as selection sort, insertion sort and merge sort. This assignment will focus on a widely used algorithm called quick sort. Just like merge sort, quick sort is a divide and conquer algorithm that was invented by British computer scientist Tony Hoare in 1959. Since the algorithm does not need any extra space to produce an output quick sort is labeled as an in place sorting algorithm whereas merge sort is an out of place sorting algorithm. The program was written in Java in collaboration with Axel Lystam at KTH.

Task 1: Quick sort in an array

Task 1 was to apply the quick sort algorithm on an array of length n. The basic idea of quick sort is to pick a pivot element, this element can be picked at random as it does not really matter. However it has been shown that picking the pivot at random has a slightly better performance than setting it as, lets say, the last index. After choosing a pivot element the algorithm partitions the elements into two sub-arrays. Lesser numbers are placed to the left of our pivot and greater numbers are placed to the right. Through recursion the sub arrays are then sorted.

```
public static void sortArray(int[] array)
    sortArray(array, 0, array.length-1);

public static void sortArray(int[] array, int lowIndex, int highIndex){
// Array completely sorted, therefor return.
    if (lowIndex >= highIndex)
        return;

    int pivotIndex = new Random().nextInt(highIndex - lowIndex) + lowIndex;
    int pivot = array[pivotIndex];
```

```
        swap(array, pivotIndex, highIndex);

        int lp = partition(array, lowIndex, highIndex, pivot);
         sortArray(array, lowIndex, lp -1);
         sortArray(array, lp + 1, highIndex);
    }

    private static void swap(int[] array, int index1, int index2){
        int temp = array[index1];
        array[index1] = array[index2];
        array[index2] = temp;
    }

    private static int partition(int[] array, int lowIndex,
                                 int highIndex, int pivot){
        int lp = lowIndex;
        int rp = highIndex;

        while(lp < rp){
           while(array[lp] <= pivot && lp < rp)
               lp++;
           while(array[rp] >= pivot && rp > lp)
               rp--;
           swap(array, lp, rp);
        }
        // Corner case, where last value could potentially be out of order.
        if(array[lp] > array[highIndex])
            swap(array, lp, highIndex);
        else
            lp = highIndex;
        return lp;
    }
```

Quick sort for an array.

A pivot is chosen at a random index, and then placed at the end of the array. Two pointers, lp and rp are then placed to point to the first and last element of the array. Both pointers traverse the list looking for elements to swap between each other. Once done, we swap the pivot element to point to the same index as our lp and rp pointers. The partitioning step is now complete, a pivot element in between two sub arrays with all numbers less than the pivot to the left and all numbers greater than the pivot to the right. To sort the sub arrays we now call the sort function recursively.

Task 2: Quick sort for a linked list.

```
private static SinglyQuickSort.Node partition(SinglyQuickSort.Node first,
SinglyQuickSort.Node last){
   if(first == last || first == null || last == null )
       return first;

       SinglyQuickSort.Node beforePivot = first;
       SinglyQuickSort.Node current = first;
       int pivot = last.value;

   while(first != last){
       if(first.value < pivot){
           beforePivot = current;
           int temp = current.value;
           current.value = first.value;
           first.value = temp;
           current = current.next;
       }
       first = first.next;
   }

   int temp = current.value;
   current.value = pivot;
   last.value = temp;

   return beforePivot;
}
public static void sort(SinglyQuickSort.Node first,
                        SinglyQuickSort.Node last){
   if(first == null || first == last || first == last.next)
       return;

   SinglyQuickSort.Node beforePivot = partition(first, last);
   sort(first, beforePivot);

   if(beforePivot != null && beforePivot == first)
       sort(beforePivot.next, last);

   else if(beforePivot != null && beforePivot.next != null)
       sort(beforePivot.next.next, last);
}
```

Implementation quick sort in an singly linked list.

Implementation of quick sort in a linked list follows the same steps as before.

- Set a pivot element.
- Partition.
- Sort through recursion.

The pivot is set to the value of the last node in our list. Since it is a linked list structure we have to declare a few more pointers to have access to the previous node when sorting the list.

Task 3: Benchmark

n	array (µs)	linked list (µs)
10	9	9
100	4	3
1000	44	44
10.000	570	575
100.000	6900	7100
1.000.000	81000	85000
10.000.000	940000	1000000

Table 1: Benchmark for quick sort on an array and a linked list of the length of n. Measured in micro seconds.

.

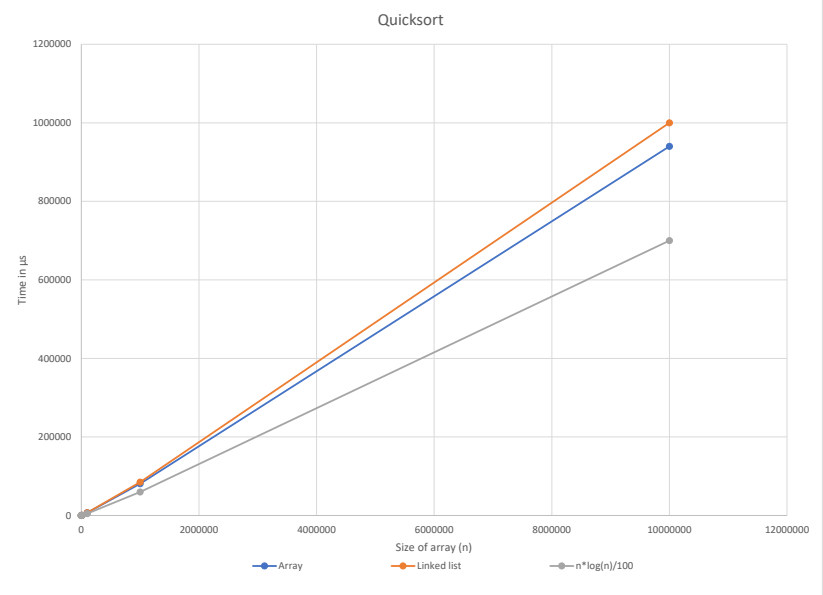

Figure 1: Benchmark for quick sort on an array and a linked list of the length of n. Measured in micro seconds.

The benchmark tests were done using a randomized sequence on the two data structures. Loop count was set to 1000 and an average time was calculated by running the benchmark several times. In table 1 we can see our two implementations described in a graph. The third function, $1/100 * nlog(n)$ is described in gray. $1/100$ can be viewed as a constant. For each measurement we tenfold our n. The two functions follow the same tenfold increase, plus a little more, with the exception of the first measurement where n is set to 10. Therefor we can conclude that both implementations of quick sort has a time complexity of $nlog(n)$ for average and best case. Worst case occurs when the pivot element is picked as an extreme value (either smallest or largest). This will only happen when the sequence is already sorted or sorted in reversed order and either last or first element is picked as our pivot element.

```
for ( int n : sizes) {
        int loop = 100;
        int[] array = {};
        LinkedList ll = new LinkedList();

        long t0 = 0;
```

```
double t1 = 0;
long t2 = 0;
double t3 = 0;

for (int i = 0; i < loop; i++) {
    array = unsorted(n);
    ll.addArray(array);

    t0 = System.nanoTime();
    QuicksortTest.quicksort(array, 0, array.length - 1);
    t1 += System.nanoTime() - t0;

    t2 = System.nanoTime();
    QuickSortLinked.quickSortLinked(ll.head, ll.tail);
    t3 += System.nanoTime() - t2;
    }
}
```

Benchmark of quick sort in the array implementation.

The benchmark which was used can be seen above. For both data structures we use the same randomized sequence. The sequence is recreated each loop iteration. The result is calculated by dividing the t total variables (t1 and t3) with the number of loops.