



SORTING TASK

[Document subtitle]



IBTEHAL AMRO

Link to the video : <https://www.youtube.com/watch?v=zQf8hT-kaOA>

The aim of this report is to demonstrate the steps I used to optimize the sorting of 500000 int number, the process has multiple steps of improvements to reach the result.

First Step

Developing process started by running the `Collections.sort(list)` on the list to examine the required time to sort this set, results shown in bellow image:

```
Result: 26.778 ±(99.9%) 2.492 s/op [Average]
Statistics: (min, avg, max) = (25.098, 26.778, 30.446), stdev = 1.648
Confidence interval (99.9%): [24.286, 29.269]

# Run complete. Total time: 00:06:04

Benchmark                Mode  Samples   Score   Score error   Units
c.b.MyBenchmark.compete  avgt      10    26.778      2.492    s/op
```

Figure 1: Collections.sort(list)

Now as image show it took around 27 seconds to sort 500000 int numbers using the `Collections.sort()` method from the java library.

So, my next goal was to beat this number to achieve better performance.

Trail one

I started by testing some of famous sorting algorithms to figure out their behavior against the number set

Note: other algorithms were tested but quick sort and merge sort work is shown bellow

1- Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Merge sort recursively calls itself on halved portions of the initial collection. With an array **A** of *n* elements. The algorithm sort the elements in 3 steps:

- 1- If **A** Contains zero or one elements then it is already sorted, otherwise, Divide the array **A** into two sub-arrays of equal number of elements.
- 2- Sort the two sub-arrays recursively using the merge sort.
- 3- Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array using the merge function.

```

void mergeSort(List<Integer> arr, int beg, int end)
{
    if (beg<end)
    {
        int mid = (beg+end)/2;
        mergeSort(arr, beg, mid);
        mergeSort(arr, beg: mid+1, end);
        merge(arr, beg, mid, end);
    }
}

```

Merge sort has complexity of $O(n \log n)$ for average case with $O(n)$ of space complexity.

Merge Sort is an "out-of-place" sorting algorithm. What this means is that Merge Sort does not sort and store the elements in the memory addresses of the collection given to it, but instead it creates and returns a completely new collection that is the sorted version of the one provided to it.

This is an important distinction because of memory usage. For very large arrays this would be a disadvantage because the data will be duplicated, which can memory problems on some systems.

Results of benchmarking the merge sort on the number set came as follows:

```

Result: 23.811 ±(99.9%) 0.674 s/op [Average]
Statistics: (min, avg, max) = (23.405, 23.811, 24.972), stdev = 0.446
Confidence interval (99.9%): [23.137, 24.484]

# Run complete. Total time: 00:05:17

Benchmark           Mode  Samples   Score   Score error   Units
c.b.MyBenchmark.compete  avgt      10    23.811      0.674    s/op

```

As shown in the image there is no such improvement between merge sort and Collections.sort from the java library.

2- Quick sort

As merge sort does not came with the results that we are expecting I used the quick sort algorithm also to test its performance

Quicksort is a divide and conquer algorithm. With an array **A** of n elements. The algorithm sorta the elements in 4 steps:

- 1- If the array contains only one or zero elements, then the array is sorted. Otherwise, Select an element from the array. This element is called the "pivot element". For example, select the element in the middle of the array.
- 2- All elements which are smaller than the pivot element is placed in one array and all elements which are larger are placed in another array.

- 3- Sort both arrays by recursively applying Quicksort to them.
- 4- Combine the arrays.

```
static void quickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, high: pivot - 1);
        quickSort(arr, low: pivot + 1, high);
    }
}
```

Quicksort can be implemented to sort "in-place". This means that the sorting takes place in the array and that no additional array needs to be created.

Quick sort has $O(n \log n)$ complexity with $O(1)$ space complexity.

```
Result: 31.658 ±(99.9%) 1.267 s/op [Average]
Statistics: (min, avg, max) = (30.950, 31.658, 33.846), stdev = 0.838
Confidence interval (99.9%): [30.391, 32.925]
```

```
# Run complete. Total time: 00:06:54
```

Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	31.658	1.267	s/op

As seen in the benchmarking result also quick sort does not provide any improvement above `Collections.sort()`

Above results are calculated by providing the list of numbers as `ArrayList of Integers` to the sorting algorithms

Second Step

As we do not see any improvements on the benchmarking results, I tried to change the data structure used in these sorting algorithms from list of Integer objects

- `mergeSort(list,0, list.size()-1);`

to using an array of primitive int

- `mergeSort(list.stream().mapToInt(i-> (int) i).toArray(),0, list.size()-1);`

this results in a drop of the execution time to the half in the case of merge sort

```
Result: 23.811 ±(99.9%) 0.674 s/op [Average]
Statistics: (min, avg, max) = (23.405, 23.811, 24.972), stdev = 0.446
Confidence interval (99.9%): [23.137, 24.484]
```

Run complete. Total time: 00:05:17

Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	23.811	0.674	s/op

```
Result: 12.852 ±(99.9%) 1.126 s/op [Average]
Statistics: (min, avg, max) = (12.027, 12.852, 14.461), stdev = 0.745
Confidence interval (99.9%): [11.727, 13.978]
```

Run complete. Total time: 00:02:49

Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	12.852	1.126	s/op

And in case of quick sort

```
Result: 31.658 ±(99.9%) 1.267 s/op [Average]
Statistics: (min, avg, max) = (30.950, 31.658, 33.846), stdev = 0.838
Confidence interval (99.9%): [30.391, 32.925]
```

Run complete. Total time: 00:06:54

Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	31.658	1.267	s/op

```
Result: 8.495 ±(99.9%) 0.238 s/op [Average]
Statistics: (min, avg, max) = (8.293, 8.495, 8.755), stdev = 0.158
Confidence interval (99.9%): [8.256, 8.733]
```

Run complete. Total time: 00:01:53

Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	8.495	0.238	s/op

That makes sense since most of the operations used in these two sorting algorithms are random access on elements and since random access on list costs $O(n)$ where in arrays it costs $O(1)$ which results in better performance to the array. also, when dealing with primitives, arrays can be faster as they avoid many boxing/unboxing conversions.

Note: After the sorting is finished the sorted array is copied back to the array.

From this point we can perform more improvements on the performance, quick sort algorithm provides better benchmarking results such that it is chosen to complete the improvement process.

Third Step

From the fact that quick sort is a divide and conquer algorithm we can take advantage of parallel processing of the separate parts of the algorithm to achieve better performance.

Parallel processing of the quick sort algorithm

At this point we conclude that converting the list of Integers into array of int primitive increased the performance of quick sort algorithm as seen in previous chapter.

Also, taking advantage of the parallel processing and the fact that quick sort algorithm can be divided into separate independent parts may increase the performance, so Fork/Join Framework has been chosen to accomplish the parallel part of the algorithm

What is the Fork/Join Framework?

The fork/join framework was designed to speed up the execution of tasks that can be divided into other smaller subtasks, executing them in parallel and then combining their results to get one.

For this reason, the subtasks must be independent of each other. By applying a divide and conquer principle; the framework recursively divides the task into smaller subtasks until a given threshold is reached. This is the fork part.

Then, the subtasks are processed independently and if they return a result, all the results are recursively combined into a single result. This is the join part.

In our case the quick sort algorithm has to be modified such that we need to identify the subtasks of the algorithm that need to be processed in parallel.

below is the modified version of the quick sort algorithm using Fork/Join Framework

```

public class QuickSortAction extends RecursiveAction {
    private int[] data;
    private int low;
    private int high;
    public QuickSortAction(int[] data) {
        this.data = data;
        low = 0;
        high = data.length - 1;
    }
    public QuickSortAction(int[] data, int low, int high) {
        this.data = data;
        this.low = low;
        this.high = high;
    }
    @Override
    protected void compute() {
        if (low < high) {
            int pivot = partition(data, low, high);
            invokeAll(new QuickSortAction(data, low, pivot), new QuickSortAction(data, pivot + 1, high));
        }
        private int partition(int[] arr, int low, int high) { // code available in the .java
file        }

        private void swap(int[] array, int l, int h) { // code available in the .java file        }
    }
}

```

use below code in the sort method to call the modified version

```


int[] sortedArray = list.stream().mapToInt(i -> (int) i).toArray();
ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
pool.invoke(new QuickSortAction(sortedArray));
pool.shutdown();
list = Arrays.asList(sortedArray); //notice that we copied the sorted array back to the list

```

this version of code results in improvement of 2 seconds from the normal quick sort

```
Result: 6.583 ±(99.9%) 0.343 s/op [Average]
Statistics: (min, avg, max) = (6.154, 6.583, 6.943), stdev = 0.227
Confidence interval (99.9%): [6.239, 6.926]

# Run complete. Total time: 00:01:31
```

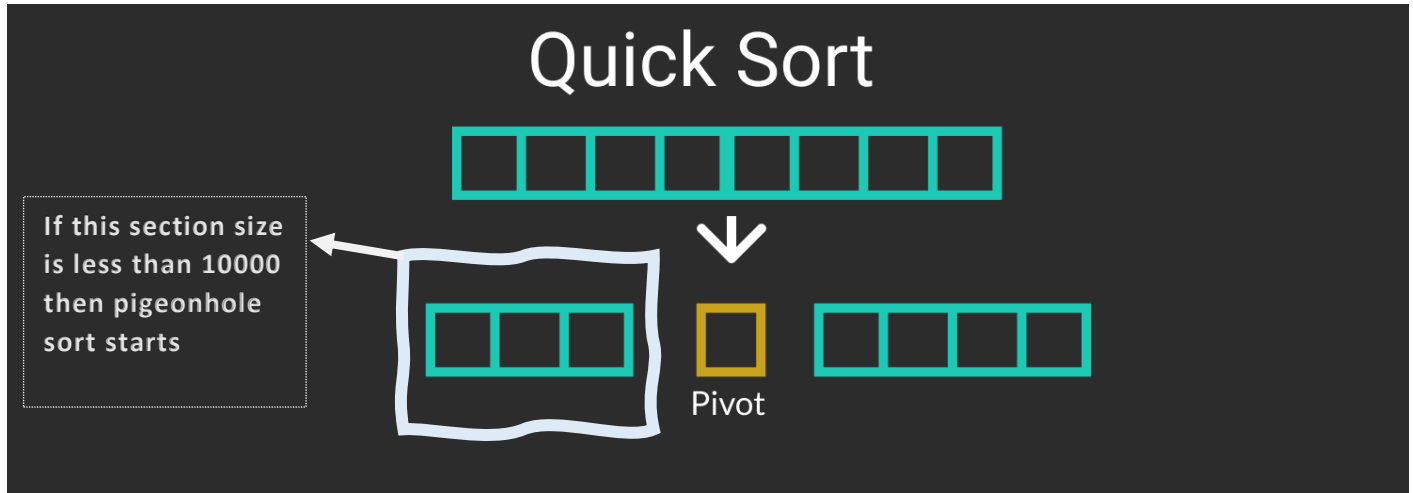


Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	6.583	0.343	s/op

at this point we converted the list to array of primitive int then used Fork/Join Framework to reach 75% of improvement from the `Collection.sort()` method.

Fourth Step

The last step of improvement was by sorting the parts of the quick sort array that are less than 10000 elements in time complexity close to linear. Some algorithms were tested and the one that has been chosen is pigeonhole sort with time complexity of $O(n+N)$.



following code snippet sorting the parts that have size less than 10000 of the quick sort array using the pigeonhole sort


```
@Override
protected void compute() {
    if (low < high) {
        if (high - low < 10000) {
            int[] copy = Arrays.copyOfRange(data, low, high + 1);
            pigeonhole_sorting(copy, copy.length, low);
        } else {
            int pivot = partition(data, low, high);
            invokeAll(new QuickSortAction(data, low, pivot),
                new QuickSortAction(data, pivot + 1, high));
        }
    }
}
```

Note: see index 1 for the pigeonhole sort code

Using the quick sort alongside with pigeonhole sort results in the following benchmarking results

```
Result: 4.533 ±(99.9%) 0.161 s/op [Average]
Statistics: (min, avg, max) = (4.352, 4.533, 4.722), stdev = 0.107
Confidence interval (99.9%): [4.371, 4.694]
```

Run complete. Total time: 00:01:03



Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	4.533	0.161	s/op

Which leads to 83% of improvement from the collections.sort() method.

Conclusion

- 1- convert the list of Integers to Array of primitive int
- 2- use quick sort to partition the list into two part
- 3- for each part if its size is less than 10000 then use the pigeonhole sort otherwise go to step 2
- 4- once the array is sorted copy the array to the list of Integers

Note: comments and long unnecessary codes were omitted to keep clean report

Index 1: Pigeonhole sort code

```
private void pigeonhole_sorting(int[] arr, int size, int low) {
    int range, i, j, index;
    int min = arr[0], max = arr[0];
    //find the maximum and minimum elements
    for (int q = 0; q < size; q++) {
        if (arr[q] > max) {
            max = arr[q];
        }
        if (arr[q] < min) {
            min = arr[q];
        }
    }
    //then find the range
    range = max - min + 1;
    int[] finalArr = new int[range];
    Arrays.fill(finalArr, 0);
    //put the elements on the holes
    for (i = 0; i < size; i++) {
        //element is added at hole index= arr[i]-min
        finalArr[arr[i] - min]++;
    }
    index = low;
    //put back the elements from the finalArr into the original array
    for (j = 0; j < range; j++) {
        while (finalArr[j]-- > 0) {
            data[index++] = j + min; // this edits the original array
        }
    }
}
```