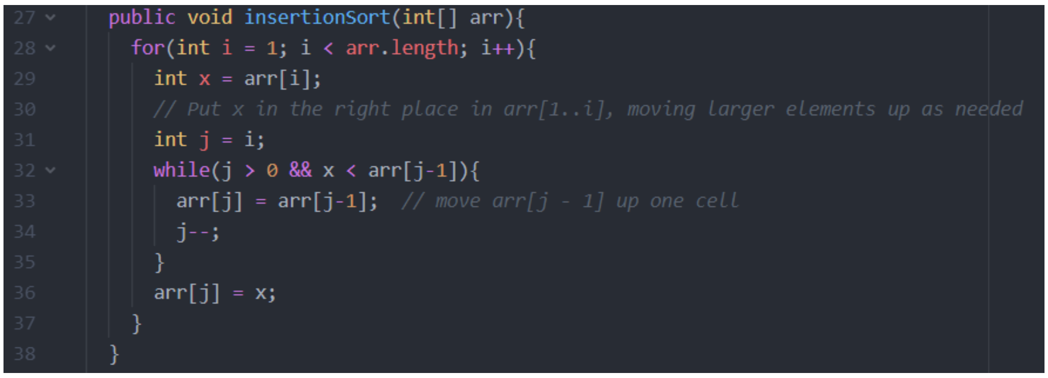
IN2010 Assignment 3

Danielius Kocan (30.10.2019)

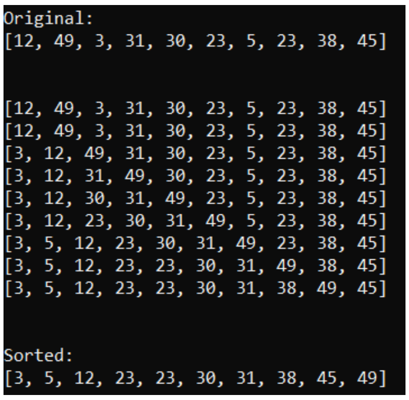
Insertion Sort



```java
public void insertionSort(int[] arr){
  for(int i = 1; i < arr.length; i++){
    int x = arr[i];
    // Put x in the right place in arr[1..i], moving larger elements up as needed
    int j = i;
    while(j > 0 && x < arr[j-1]){
      arr[j] = arr[j-1];  // move arr[j - 1] up one cell
      j--;
    }
    arr[j] = x;
  }
}
```

In this algorithm I had to change int i = 2 to int i = 1 in first for loop. I also had to change j>1 to j>0. Because sorting algorithm from the book wasn't sorting the first number in array.

Testing correctness and pattern:

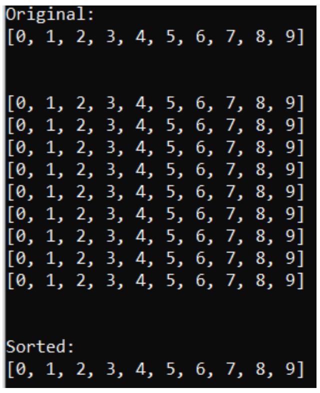


```
Original:
[12, 49, 3, 31, 30, 23, 5, 23, 38, 45]

[12, 49, 3, 31, 30, 23, 5, 23, 38, 45]
[12, 49, 3, 31, 30, 23, 5, 23, 38, 45]
[3, 12, 49, 31, 30, 23, 5, 23, 38, 45]
[3, 12, 31, 49, 30, 23, 5, 23, 38, 45]
[3, 12, 30, 31, 49, 23, 5, 23, 38, 45]
[3, 12, 23, 30, 31, 49, 5, 23, 38, 45]
[3, 5, 12, 23, 30, 31, 49, 23, 38, 45]
[3, 5, 12, 23, 23, 30, 31, 49, 38, 45]
[3, 5, 12, 23, 23, 30, 31, 38, 49, 45]

Sorted:
[3, 5, 12, 23, 23, 30, 31, 38, 45, 49]
```

Random input array:

As we can see we doesn't get lowest number on first index in array from the start, because this algorithm goes throw whole array and sort it on the way.

Algorithm takes number on index 1. And after that checks if numbers from left side are bigger than current. And does it while it comes to index or if number for the left side is lower than current number algorithm working with. So, in case we know that if lowest number at the end of original array, it will come to the index 0 in sorted array only at the end. Because this algorithm works with on by one number while it goes throw whole original array.



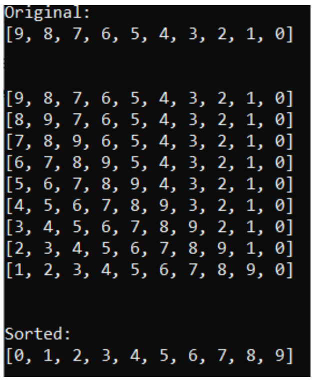
```
Original:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Array that is sorted by rising values:

Here we can see what algorithm does nothing. Because algorithm checks if number from the left side to a current number is bigger than current number. In this case sorting time is much faster, O(n).

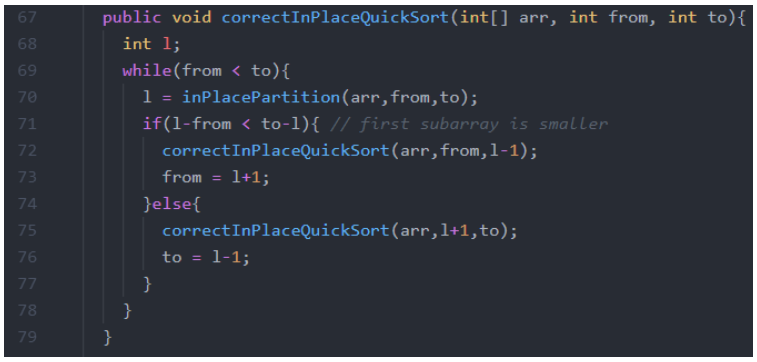
Array that is sorted by falling value

Here we can see again, same as in random array. Algorithm doesn't find lowest number at first, but goes throw array and sort it "on the way to the end".

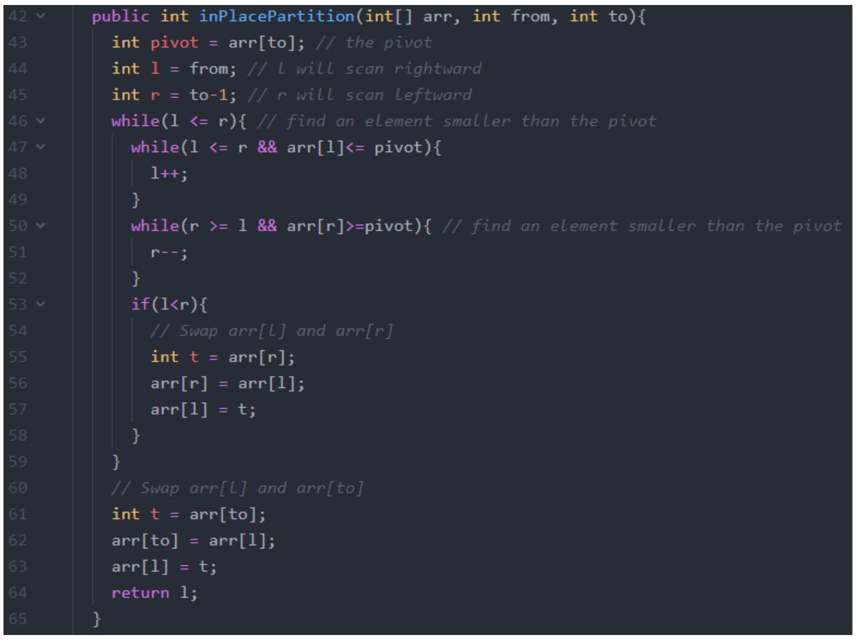


```
Original:
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]


[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[8, 9, 7, 6, 5, 4, 3, 2, 1, 0]
[7, 8, 9, 6, 5, 4, 3, 2, 1, 0]
[6, 7, 8, 9, 5, 4, 3, 2, 1, 0]
[5, 6, 7, 8, 9, 4, 3, 2, 1, 0]
[4, 5, 6, 7, 8, 9, 3, 2, 1, 0]
[3, 4, 5, 6, 7, 8, 9, 2, 1, 0]
[2, 3, 4, 5, 6, 7, 8, 9, 1, 0]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]


Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Quick Sort



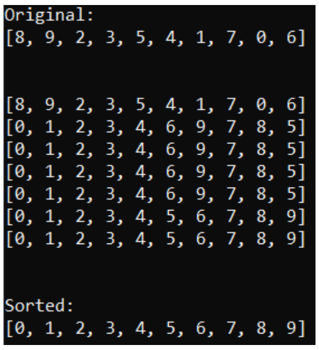
```java
public void correctInPlaceQuickSort(int[] arr, int from, int to){
  int l;
  while(from < to){
    l = inPlacePartition(arr,from,to);
    if(l-from < to-l){ // first subarray is smaller
      correctInPlaceQuickSort(arr,from,l-1);
      from = l+1;
    }else{
      correctInPlaceQuickSort(arr,l+1,to);
      to = l-1;
    }
  }
}
```



```java
public int inPlacePartition(int[] arr, int from, int to){
  int pivot = arr[to]; // the pivot
  int l = from; // l will scan rightward
  int r = to-1; // r will scan leftward
  while(l <= r){ // find an element smaller than the pivot
    while(l <= r && arr[l]<= pivot){
      l++;
    }
    while(r >= l && arr[r]>=pivot){ // find an element smaller than the pivot
      r--;
    }
    if(l<r){
      // Swap arr[l] and arr[r]
      int t = arr[r];
      arr[r] = arr[l];
      arr[l] = t;
    }
  }
  // Swap arr[l] and arr[to]
  int t = arr[to];
  arr[to] = arr[l];
  arr[l] = t;
  return l;
}
```

I did almost no changes when I was trying to write code in java from pseudo code in the book. It was hard to fully understand how code works. But after some small examples on paper, the code gave much more meaning.

Testing correctness and pattern:



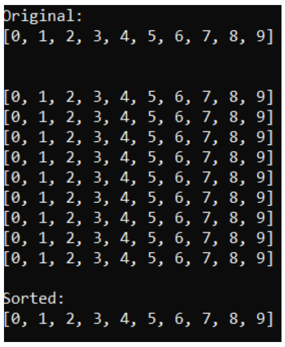
```
Original:
[8, 9, 2, 3, 5, 4, 1, 7, 0, 6]


[8, 9, 2, 3, 5, 4, 1, 7, 0, 6]
[0, 1, 2, 3, 4, 6, 9, 7, 8, 5]
[0, 1, 2, 3, 4, 6, 9, 7, 8, 5]
[0, 1, 2, 3, 4, 6, 9, 7, 8, 5]
[0, 1, 2, 3, 4, 6, 9, 7, 8, 5]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Random input array:

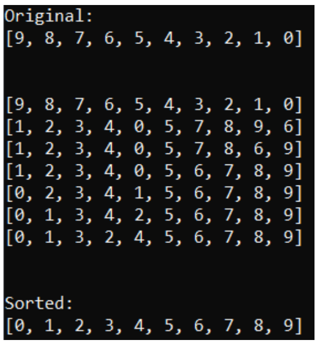
I can't really see any pattern in this sorting. My algorithm choosing random pivot, so it's kind of hard to follow algorithm sorting.

Array that is sorted by rising values:

Array doesn't change, in this case running time must be a bit lower than with random input. Because when we choose random pivot, algorithm will always find out that he doesn't need to swap any numbers in array. So, algorithm will run relatively faster than usual.



```
Original:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



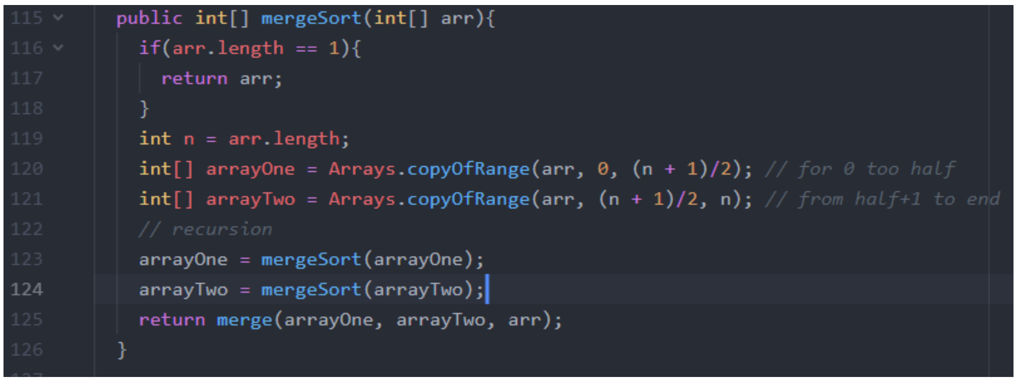
```
Original:
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]


[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[1, 2, 3, 4, 0, 5, 7, 8, 9, 6]
[1, 2, 3, 4, 0, 5, 7, 8, 6, 9]
[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]
[0, 2, 3, 4, 1, 5, 6, 7, 8, 9]
[0, 1, 3, 4, 2, 5, 6, 7, 8, 9]
[0, 1, 3, 2, 4, 5, 6, 7, 8, 9]


Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Array that is sorted by falling value

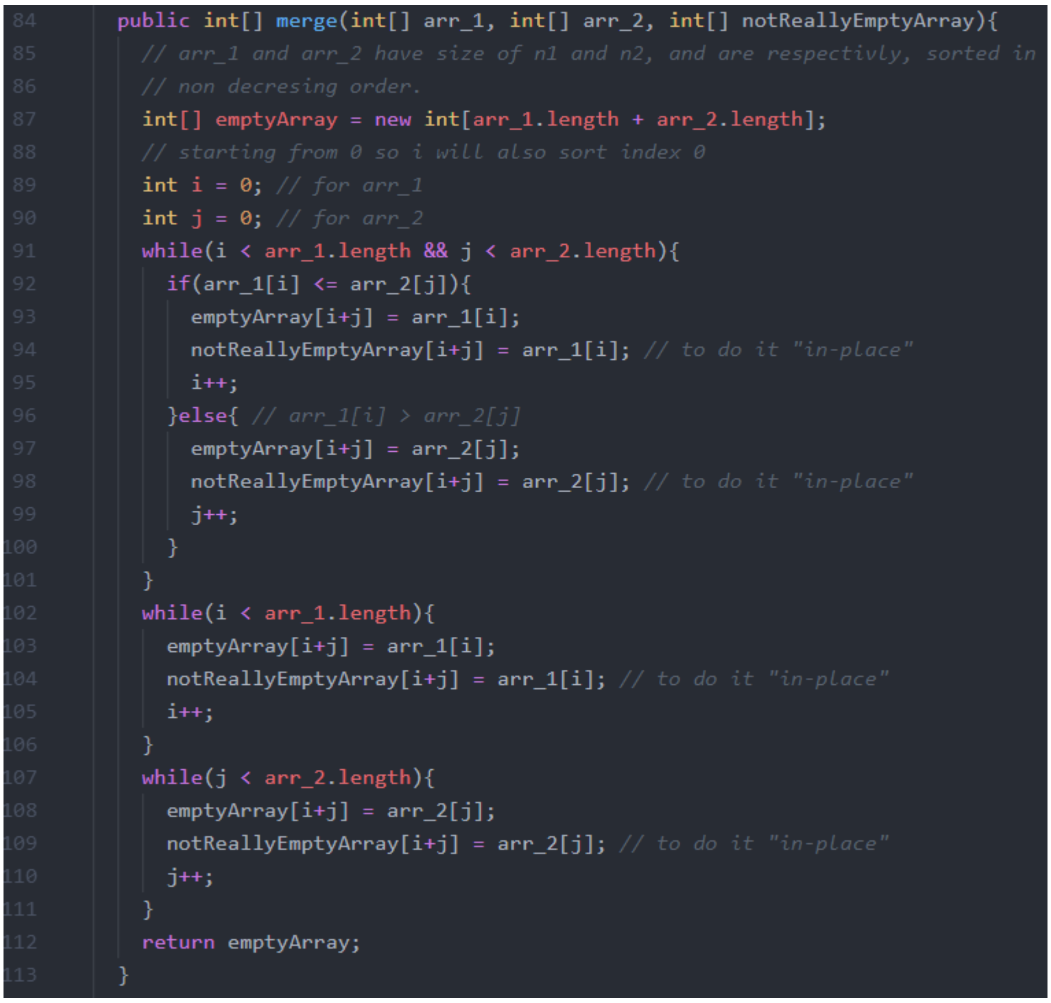
Same as in random input array, algorithm chooses random pivot, and checks if left number is bigger than pivot and if right number is lower than pivot, so if it a case so those numbers change places. After few shifts, algorithm splits array on two parts, and takes one random pivot in each part, and does the same as with full array. After few shifts and splits, our array will be sorted.

From the example I assume that first pivot was 5. Because numbers 1,2,3,4,0 were shifted in left side, while numbers 7,8,9,6 were shifted in right side. (Because of how algorithm works). So after that algorithm found one pivot in [1,2,3,4,0] and one pivot in [7,8,9,6] and sorted them.

Merge Sort



```java
public int[] mergeSort(int[] arr){
  if(arr.length == 1){
    return arr;
  }
  int n = arr.length;
  int[] arrayOne = Arrays.copyOfRange(arr, 0, (n + 1)/2); // for 0 too half
  int[] arrayTwo = Arrays.copyOfRange(arr, (n + 1)/2, n); // from half+1 to end
  // recursion
  arrayOne = mergeSort(arrayOne);
  arrayTwo = mergeSort(arrayTwo);
  return merge(arrayOne, arrayTwo, arr);
}
```

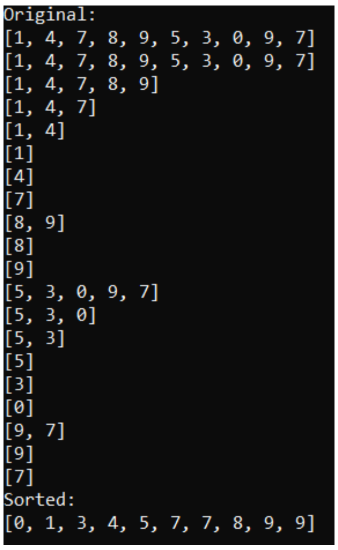


```java
public int[] merge(int[] arr_1, int[] arr_2, int[] notReallyEmptyArray){
  // arr_1 and arr_2 have size of n1 and n2, and are respectivly, sorted in
  // non decresing order.
  int[] emptyArray = new int[arr_1.length + arr_2.length];
  // starting from 0 so i will also sort index 0
  int i = 0; // for arr_1
  int j = 0; // for arr_2
  while(i < arr_1.length && j < arr_2.length){
    if(arr_1[i] <= arr_2[j]){
      emptyArray[i+j] = arr_1[i];
      notReallyEmptyArray[i+j] = arr_1[i]; // to do it "in-place"
      i++;
    }else{ // arr_1[i] > arr_2[j]
      emptyArray[i+j] = arr_2[j];
      notReallyEmptyArray[i+j] = arr_2[j]; // to do it "in-place"
      j++;
    }
  }
  while(i < arr_1.length){
    emptyArray[i+j] = arr_1[i];
    notReallyEmptyArray[i+j] = arr_1[i]; // to do it "in-place"
    i++;
  }
  while(j < arr_2.length){
    emptyArray[i+j] = arr_2[j];
    notReallyEmptyArray[i+j] = arr_2[j]; // to do it "in-place"
    j++;
  }
  return emptyArray;
}
```

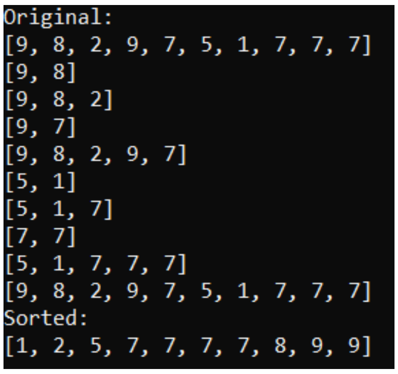
To make pseudo code work I had to make sure that this sorting algorithm starts from index 0. Because pseudo code in the book started from index 1 and never sorted the first and last element in array. But in my implementation merge sort sorts every element in array. Also, I had to make a recursion function that will split array to small pieces and build it up again by using merge method. I didn't find any pseudo code for that in the book, that's why I had to spend some time to find solution for that.

Testing correctness and pattern:

Random input array:



```
Original:
[1, 4, 7, 8, 9, 5, 3, 0, 9, 7]
[1, 4, 7, 8, 9, 5, 3, 0, 9, 7]
[1, 4, 7, 8, 9]
[1, 4, 7]
[1, 4]
[1]
[4]
[7]
[8, 9]
[8]
[9]
[5, 3, 0, 9, 7]
[5, 3, 0]
[5, 3]
[5]
[3]
[0]
[9, 7]
[9]
[7]
Sorted:
[0, 1, 3, 4, 5, 7, 7, 8, 9, 9]
```



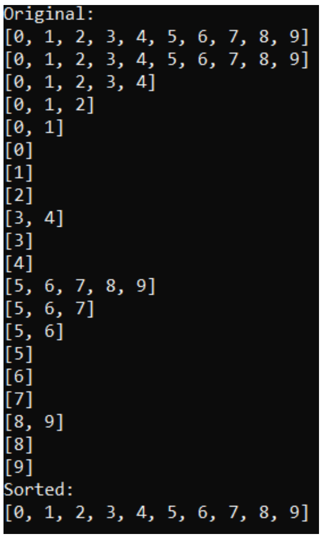
```
Original:
[9, 8, 2, 9, 7, 5, 1, 7, 7, 7]
[9, 8]
[9, 8, 2]
[9, 7]
[9, 8, 2, 9, 7]
[5, 1]
[5, 1, 7]
[7, 7]
[5, 1, 7, 7, 7]
[9, 8, 2, 9, 7, 5, 1, 7, 7, 7]
Sorted:
[1, 2, 5, 7, 7, 7, 7, 8, 9, 9]
```

It was hard to find the right spot for showing the whole array at given time. Because my algorithm was spliting whole array and building it up again. (With help of recursion). So most of the time I got splited array, that gets build up back again at the end of recursion.

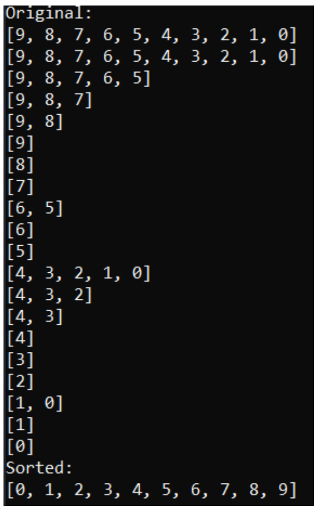
So on the image can we see that array being splited in two parths. And after that algorithm use recursion to split first part on half, and so on until it comes to array with length 1. After that by using merge() method, algorithm sort it and returns sorted array back to recursion call, until recursion comes back where it been called.

Array that is sorted by rising values:

Here array was sorted from the start, but algorithm had to go throw it and split it as always, to be sure that array is sorted. So, I think running time is the same as in random generated array.



```
Original:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4]
[0, 1, 2]
[0, 1]
[0]
[1]
[2]
[3, 4]
[3]
[4]
[5, 6, 7, 8, 9]
[5, 6, 7]
[5, 6]
[5]
[6]
[7]
[8, 9]
[8]
[9]
Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



```
Original:
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 8, 7, 6, 5]
[9, 8, 7]
[9, 8]
[9]
[8]
[7]
[6, 5]
[6]
[5]
[4, 3, 2, 1, 0]
[4, 3, 2]
[4, 3]
[4]
[3]
[2]
[1, 0]
[1]
[0]
Sorted:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Array that is sorted by falling value

Here same as in random generated and sorted arrays, algorithm needs to go split whole algorithm and do the same process as always. So I think running time is almost the same as always.

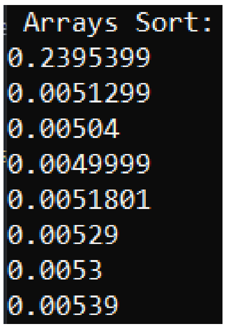
Time testing:

Number of elements in start array	Sorted Array average Time (Milliseconds)	Reversed Array average Time (Milliseconds)	Random Array average Time (Milliseconds)
1 000 (Arrays.sort)	0.001633	0.00172598	0.003755
Insertion Sort	1.7201E-4	0.01741597	0.043939
Quick Sort	0.005155	0.0055740	0.0432320
Merge Sort	0.0329389	0.01082298	0.023024
5 000 (Arrays.sort)	0.001601	0.00569100	0.015749
Insertion Sort	7.5201E-4	0.41072802	0.258246
Quick Sort	0.0266670	0.02718399	0.087524
Merge Sort	0.0706009	0.04461502	0.067112
10 000 (Arrays.sort)	0.0056610	0.00294398	0.028251
Insertion Sort	0.0014780	1.63154300	0.886062
Quick Sort	0.050483	0.05415500	0.140393
Merge Sort	0.1187210	0.09637099	0.125809
50 000 (Arrays.sort)	0.0066809	0.06269597	0.084904
Insertion Sort	0.0075129	40.6474089	19.97602
Quick Sort	0.3131499	0.31908900	0.658426
Merge Sort	0.2715970	0.25035602	0.507316
100 000 (Arrays.sort)	0.0066979	0.00722599	0.111102
Insertion Sort	0.0143660	170.493650	80.51251
Quick Sort	0.506807	0.62179299	1.325792
Merge Sort	0.5618540	0.66615201	0.786083
500 000 (Arrays.sort)	0.0283320	0.02786296	0.358092
Insertion Sort	0.0750229	4388.53409	2168.214
Quick Sort	2.7961960	3.1087170	6.435503
Merge Sort	3.5721600	3.53948495	3.560829
1 000 000 (Arrays.sort)	0.0583760	0.05645005	0.7550129
Insertion Sort	0.1498229	17799.2801	8776.5755
Quick Sort	5.5170259	5.863928	13.644944
Merge Sort	7.7513460	8.03499499	7.3687890

From what I can see, there is only one surprise. I wasn't thinking that Arrays.sort can be so fast. Before running test, I was thinking that merge sort will be the fastest. There are only few places where insertion sort is a bit faster than Arrays.sort, and this is only with small arrays, and only sorted arrays. And that's because if array is sorted, insertion algorithm will have running time O(n), because this is algorithms best case. With bigger arrays, insertion sort takes second place by sorting speed in sorted arrays, but does really bad work in random arrays, and even worse in reverse sorted array, because running time for insertion sort in normal/worse situation is O(n^2).

The merge sort algorithm has the most stable performance in all three categories. Because merge sorting algorithm have running time O(nlog(n)) all the time. In other side we have quick sort that often have running time of O(nlog(n)), however worst case for quick sort can be O(n^2). And we can see that at the table with average running time. As we can see there is some longer running time in some cases for quick sort. While merge sort holds almost the same time in every case. That's why often merge sort is faster. However quick sort needs less additional storage space to work.

While I was testing algorithm time, my program was checking same algorithm 10 times with 10 different arrays to find average running time. Because of that I found out that with small arrays few first runs have longer running time. At first, I was thinking that there is a problem in my algorithm code. It looks like that ⇨⇨⇨⇨



```
 Arrays Sort:
0.2395399
0.0051299
0.00504
0.0049999
0.0051801
0.00529
0.0053
0.00539
```

https://stackoverflow.com/questions/35960667/sorting-second-time-round-is-faster

But after some research on internet I found out. That there are many things that affect running time of few first tests. The first run's long runtime is probably explained by JIT (just-in-time) compilation. The first run is getting the disadvantage of running in the interpreter for a while plus the extra costs of compilation. This is why "warming up" is a common term in Java benchmarks.