Lecture 14

Advanced Sorting:
MergeSort and QuickSort

Last time in relation to sorting:

In lecture 8, we looked at three simple sorting algorithms: Bubble Sort, Selection Sort, and Insertion Sort.

There are basically two steps involved in these sorting algorithms.

- Compare two items
- Swap two items or copy (shift) items

However, each of them has a different invariant, the condition that remains unchanged as the algorithm proceeds.

- Bubble Sort: values after “out” variable are sorted. (Right-hand side)
- Selection Sort: values less than “out” variable are sorted. (Left-hand side)
- Insertion Sort: At the end of each round, values less than “out” variable are PARTIALLY sorted. (Left-hand side but partially)

In practice, insertion sort could run faster than the other two because it requires less number of comparisons on average. In fact, if the input array is already sorted, insertion sort’s running time complexity is linear. However, these three have the same worst-case running time complexity of O(n^2).

We cannot just sit, satisfied with the insertion sort. There must be a better way!

Today, we will look into some other sorting algorithms that might run faster than these.

Merge Sort

Merge Sort is a great example of divide and conquer algorithms:

- Divide the given problem into simpler versions of itself.
- Conquer each problem using the same process.
- Finally, combine the results of simpler ones to have the final answer.

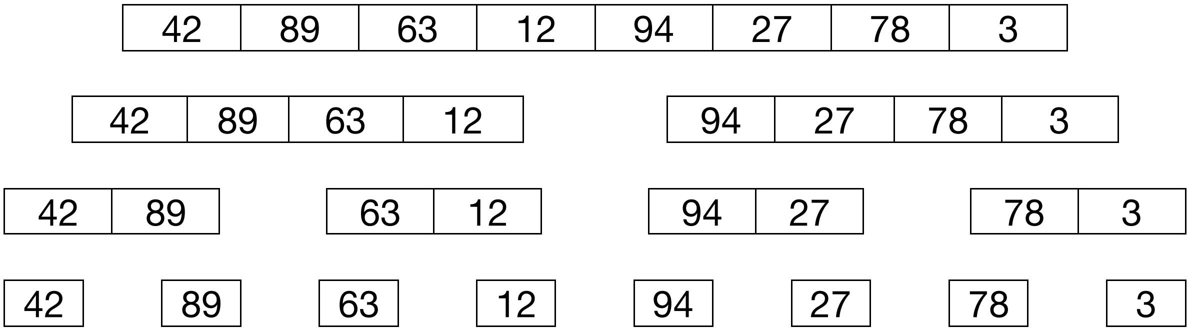
Step 1: Conceptual View

There are three steps involved in merge sort.

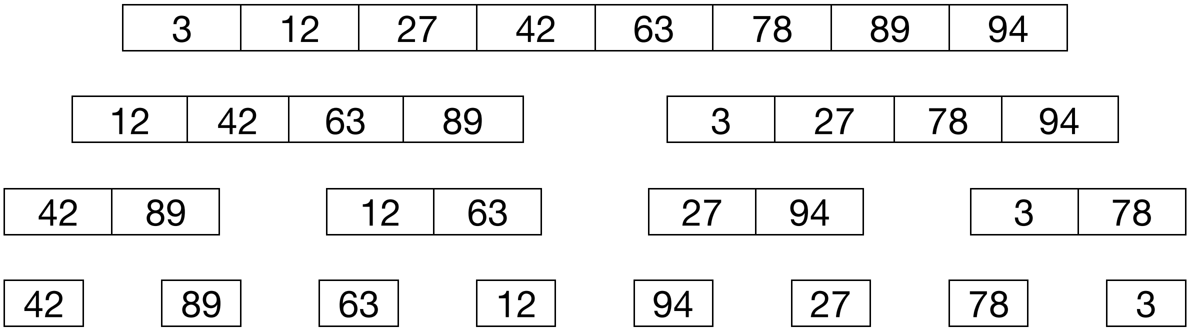
- Sort the first half using merge sort.
- Sort the second half using merge sort.
- Merge the sorted two halves to create the final sorted result.

What do you see here? Do you see recursions happening here?

Dividing processes (from top to bottom)



Merging processes (from bottom to top)



The key algorithm of merge sort is ***merging***.

Let’s take a closer look at the merging process of the last step in the previous example.

Array a				*Array b*				*Array c*							
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
$\underline{12}$	42	63	89	$\underline{3}$	27	78	94	3							
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
$\underline{12}$	42	63	89	3	$\underline{27}$	78	94	3	12						
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	$\underline{42}$	63	89	3	$\underline{27}$	78	94	3	12	27					
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	$\underline{42}$	63	89	3	27	$\underline{78}$	94	3	12	27	42				
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	42	$\underline{63}$	89	3	27	$\underline{78}$	94	3	12	27	42	63			
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	42	63	$\underline{89}$	3	27	$\underline{78}$	94	3	12	27	42	63	78		
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	42	63	$\underline{89}$	3	27	78	$\underline{94}$	3	12	27	42	63	78	89	
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	42	63	89	3	27	78	$\underline{94}$	3	12	27	42	63	78	89	94

This is a snapshot of the final merging process. Remember! This happens recursively!

Step 2: Implementation View

Since *merging* is the crux of the merge sort algorithm, let’s try to implement merge() method first.

```
/**
 * Merges two sorted arrays into a new sorted array
 * Once the merge is done, return the merged array.
 */
private static int[] merge(int[] a, int[] b) {
    // Create a new array
    int[] merged = new int[__________________________];
    // initialize all of the indices
    int indexA = 0, indexB = 0, indexM = 0;
    // add correct values to the new array
    while (indexA < ____________ && indexB < ______________) {
        if (a[indexA] < b[indexB]) {
            merged[indexM] = a[___________________];
            indexA = __________________________;
        } else {
            merged[indexM] = b[___________________];
            indexB = __________________________;
        }
        indexM = _________________________;
    }
    return merged;
}
```

Would this work properly?
If not, can you find what the issue is?

What would be the output of the following code using the merge method above?

```
int[] a = {12, 42, 63, 89};
int[] b = {3, 27, 78, 94};
System.out.println(Arrays.toString(merge(a,b)));
```

It is time for you to fix the merge method!

```
/**
 * Merges two sorted arrays into a sorted new array
 * Once the merge is done, return the merged array.
 */
private static int[] merge(int[] a, int[] b) {
    // Create a new array
    int[] merged = new int[__________________________];
    // initialize all of the indices
    int indexA = 0, indexB = 0, indexM = 0;
    // add correct values to the new array
    while (indexA < ____________ && indexB < ______________) {
        if (a[indexA] < b[indexB]) {
            merged[indexM] = a[___________________];
            indexA = __________________________;
        } else {
            merged[indexM] = b[___________________];
            indexB = __________________________;
        }
        indexM = _________________________;
    }

    // need some additional work









    return merged;
}
```

Now that we have a working merge method, it is time to implement our mergeSort() method.

As we said, we will implement it recursively. What do we need to think about first?

That is right. Base case!

How are we going to divide an array?
Left half should have the length of list.length/2
Right half should have the length of list.length-left.length

```
public static int[] mergeSort(int[] unsorted) {
    // base case
    if (unsorted.length <= ________) return __________________;

    int mid = unsorted.length / 2;
    // create left array
    int[] left = new int[_____________________];
    System.arraycopy(unsorted, ____, left, ____, ___________);

    // create right array
    int[] right = new int[________________________________];
    System.arraycopy(unsorted, _______, right, ____, ________);

    // call itself with the left half
    left = ____________________________;

    // call itself with the right half
    right = ___________________________;

    // merge and return the merged array
    return ________________________________________;
}
```

Efficiency of merge sort

Our focus is on the number of copies happening during the merging process.

42		89		63		12		94		27		78		3

42	89		12	63		27	94		3	78

From the 8 arrays (each with one item) to 4 arrays (each with two items), how many copies do we need to perform?

42	89		12	63		27	94		3	78

12	42	63	89		3	27	78	94

From the 4 arrays (each with two items) to 2 arrays (each with four items), how many copies do we need to perform?

12	42	63	89		3	27	78	94

3	12	27	42	63	78	89	94

From the 2 arrays (each with four items) to 1 array (with eight items), how many copies do we need to perform?

Now, how many steps or levels did we go through to have one array with sorted 8 items?

As a side note, can you figure out what is the maximum and minimum number of comparisons performed in each step?

Quick Sort

Step 1: Conceptual View

There are three basic steps involved in quick sort

- Partition the array or sub-arrays into left (smaller values) and right (larger values) groups.
- Call itself again to sort the left group.
- Call itself again to sort the right group.

Partitioning is the underlying mechanism of quicksort.

For example, I might want to divide students into two groups; those with grade point averages higher than 3.5 and those with grade point averages lower than 3.5.

The pivot value
It is the value that is used to determine which group a value is placed. So, if I try to divide students based on their GPA of 3.5, 3.5 is the pivot value.

1. Initial array that is not partitioned

42	89	63	12	94	27	78	3	50	36

2. For now, I decide to choose the last item to be the pivot value: 36

3. Conceptually, we have two sub-arrays that are partitioned.

36

4. The result of the first sort

3	27	12	36	63	94	89	78	42	50

5. Choose the last value in the left group as the pivot: 12

3	27	12

6. Conceptually, we again have two sub-arrays that are partitioned.

7. The result of this second sort

Notice that this does not necessarily divide the array in half. It all depends on the pivot value.

Now, when do we stop? Time to think more algorithmically!

Step 2: Implementation View

Since partitioning is the crux of the quick sort, let's try to implement partition() method first.

This method will partition a given array with the given pivot value

```
private static int partition(int[] arr, int left, int right, int
pivot){
    int leftPointer = _________________;
    int rightPointer = right;

    while (true) {
        while (arr[__________________________] < pivot) ;
        while (_________________ && arr[_______________] > pivot);
        if (leftPointer >= _________________________) {
            break;
        } else {
            swap(arr, leftPointer, ____________________);
        }
    }
    swap(arr, leftPointer, ________________);
    return ________________________;
}
```

Initial values and initial call

```
int[] arr = {12, 10, 18, 2, 15, 13};
int right = arr.length-1;
int pivot = arr[right];
partition(arr, 0, right, pivot);
```

Tracing initial call of partition(arr, 0, 5, 13);

	leftPointer	rightPointer	value	pivot	compared to pivot
Initial	-1	5		13	
arr[++leftPointer]	0	5	12	13	12 < 13
arr[++leftPointer]				13	
arr[++leftPointer]				13	
arr[--rightPointer]				13	
arr[--rightPointer]				13	

leftPointer : 2 < rightPointer: 3

swap(arr, leftPointer, rightPointer); → swap(arr, 2, 3);

arr is now {12, 10, 2, 18, 15, 13}

	leftPointer	rightPointer	value	Pivot	compared to pivot
Initial	2	3		13	
arr[++leftPointer]	3	3	18	13	18 > 13
arr[--rightPointer]				13	

leftPointer : 3 > rightPointer : 2 → break

swap(arr, leftPointer, right) → swap(arr, 3, 5)

arr is now {12, 10, 2, 13, 15, 18}

Now what do we see?

All of the values that are smaller than the pivot (13) are where?

All of the values that are bigger than the pivot (13) are where?

Now, how would we call this method to make the whole array sorted?

Recursive quick sort method

```
// recursive helper method
private static void quickSort(int[] unsorted, int left, int
right) {
    // base case
    if (_______________________________) {
        return;
    }
    // last value is the pivot value
    int pivot = unsorted[right];
    int partition = partition(unsorted, left, right, pivot);

    quickSort(unsorted, ________________, _________________);
    quickSort(unsorted, ________________, _________________);
}

// public method for quick sort
public static void quickSort(int[] unsorted) {
    quickSort(unsorted, ________________, _________________);
}
```

Efficiency of quick sort

1) Efficiency of the Partition Algorithm

The two pointers, leftPointer and rightPointer, start at opposite ends of the array and move toward each other. As they get closer to each other, there are stops and swaps, if necessary.

When they meet or cross, a partition is complete.

If there were twice as many items to partition, it would take twice longer.

So, the running time is proportional to ________.

2) Number of recursive calls in the worst case
Assuming that we are using the implementation here, let's take an array of {7, 6, 5, 4, 3, 2, 1}.

The following is a conceptual view.

Call	Pivot
quickSort({7, 6, 5, 4, 3, 2, 1})	1
quickSort({7, 6, 5, 4, 3, 2})	2
quickSort({7, 6, 5, 4, 3})	3
quickSort({7, 6, 5, 4})	4
quickSort({7, 6, 5})	5
quickSort({7, 6})	6
quickSort({7})	

As we can see, we need n-1 recursive calls. And, as we saw in the partitioning section, for each recursive call, we need O(k) comparisons to sort a sub-array of size k.

Do you see that this becomes quadratic running time complexity?

Now, how do we mitigate this?
Let's take a look at the same example with different choice of the pivot value.

Call	Pivot
quickSort({7, 6, 5, 4, 3, 2, 1})	4
quickSort({3,2,1}) quickSort({7,6,5})	2 and 6
quickSort({3}), quickSort({1}), quickSort({5}), quickSort({7})	

If we always pick ________________ among the elements in the sub-array, then half of the elements would be less than ___________ and the other half would be greater than _______________.

Which makes sure that we will be guaranteed to have _________ recursive calls.
But, it requires additional overhead to compute ________________ of all the elements.

What are possible solutions?

Choosing a random value to be the pivot value can be an option

Or

It is possible to find _________________________ of three elements (first, last and middle) in an array.

With this improvement, we can finally conclude that the quick sort's expected running time complexity is O(n log n)

*** Important!**

You cannot say that quick sort's worst case running time complexity is O(n log n). In fact, its worst case running time complexity is O(n^2). It is important for you to know its possibility of degeneration and strategies to mitigate it. We will explore this further in lab 5.

To sum it up!

In merge sort, dividing the input array is simple and easy but it is expensive to merge those sorted left and right sub-arrays.

In quick sort, dividing the original problem into sub-problems (partitioning) is more complicated and expensive whereas combining the results from sub-problems together is easy.