

# MERGESORT

David Kauchak  
CS 62 – Spring 2021

1

## Admin

Compression assignment

Lab tomorrow

2

## Sorting

Insertion sort

Selection sort

How do they work? Best, worst, average case runtime?

3

## Selection sort

3 44 38 5 47 1 36 26

sorted    unsorted

---

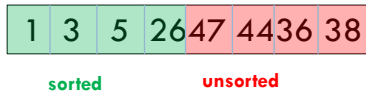
Divide the array into two parts: a sorted part on the left and an unsorted part on the right

Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

4

## Selection sort



Divide the array into two parts: a sorted part on the left and an unsorted part on the right

Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

5

## Selection sort: overall runtime

Best case = worst case = averages case =  $O(n^2)$

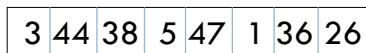
Divide the array into two parts: a sorted part on the left and an unsorted part on the right

Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

6

## Insertion sort



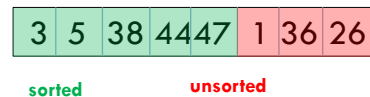
Divide the array into two parts:  
left part: left elements in sorted order  
right part: right elements in unsorted order

Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

7

## Insertion sort



Divide the array into two parts:  
left part: left elements in sorted order  
right part: right elements in unsorted order

Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

8

## Insertion sort: overall runtime

Best case:  $O(n)$ , the array is already sorted

Worst case:  $O(n^2)$ , the array is reverse sorted (same sum as before)

Average case:  $O(n^2)$ ,  $n$  iterations and still have to move  $n/2$  entries on average

Divide the array into two parts:

left part: left elements in sorted order

right part: right elements in unsorted order

Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

9

## Sorting algorithm properties

### Stable sorting algorithms

If there are ties, the elements occur in their original order

Excel demo!

10

### Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

### Insertion sort

Divide the array into two parts:

left part: left elements in sorted order

right part: right elements in unsorted order

Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

Are these stable?

11

### Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

### Insertion sort is stable

Divide the array into two parts:

left part: left elements in sorted order

right part: right elements in unsorted order

Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

12

## Sorting algorithm properties

### In-place sorting

Can be done without additional memory, i.e., another array

13

### Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

#### Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

### Insertion sort

Divide the array into two parts:

left part: left elements in sorted order

right part: right elements in unsorted order

Are these in-place?

#### Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

14

### Selection sort is in place

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

#### Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

### Insertion sort is in-place

Divide the array into two parts:

left part: left elements in sorted order

right part: right elements in unsorted order

#### Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

15

### Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

#### Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

### Insertion sort

Divide the array into two parts:

left part: left elements in sorted order

right part: right elements in unsorted order

What questions do we ask about the data?

#### Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

16

### Selection sort

Divide the array into two parts: a sorted part on the left and an unsorted part on the right

Repeat:

- Find the smallest element in the unsorted part
- Swap it with the leftmost element of the unsorted array
- The sorted array is now one element larger

### Insertion sort

Divide the array into two parts:  
left part: left elements in sorted order  
right part: right elements in unsorted order

Repeat:

- Look at the next element in the unsorted part
- Find the correct location in the sorted part (by sliding each item right one at a time)
- The sorted array is now one element larger

What questions do we ask about the data?

Compare to other elements

17

## Comparable interface

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

### Interface Comparable<T>

int compareTo(T other)

- -1: this object is less than other (technically, any negative number)
- 0: this object is equal to other
- 1: this object is greater than other (technically, any positive number)

18

## Which algorithm is this?

```
public static void sort(Comparable[] a) {
    for( int i = 0; i < a.length; i++ ) {
        int smallestIndex = i;
        for( int j = i+1; j < a.length; j++ ) {
            if( a[j].compareTo(a[smallestIndex]) < 0 ) {
                smallestIndex = j;
            }
        }
        Comparable temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

19

## Which algorithm is this?

```
public static void sort(Comparable[] a) {
    for( int i = 0; i < a.length; i++ ) {
        int smallestIndex = i;
        for( int j = i+1; j < a.length; j++ ) {
            if( a[j].compareTo(a[smallestIndex]) < 0 ) {
                smallestIndex = j;
            }
        }
        Comparable temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

20

## Which algorithm is this?

```
public static void sort(Comparable[] a) {
    for( int i = 0; i < a.length; i++) {
        int smallestIndex = i;
        for( int j = i+1; j < a.length; j++) {
            if( a[j].compareTo(a[smallestIndex]) < 0 ) {
                smallestIndex = j;
            }
        }
        Comparable temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

find the smallest value in the unsorted part (i+1... end)

21

## Which algorithm is this?

```
public static void sort(Comparable[] a) {
    for( int i = 0; i < a.length; i++) {
        int smallestIndex = i;
        for( int j = i+1; j < a.length; j++) {
            if( a[j].compareTo(a[smallestIndex]) < 0 ) {
                smallestIndex = j;
            }
        }
        Comparable temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

swap i and the smallest value

22

## A better way

We can constrain the type variable to only allow for classes that implement Comparable<E>.

```
public static <E extends Comparable<E>> void sortBetter(E[] a) {
    for( int i = 0; i < a.length; i++) {
        int smallestIndex = i;
        for( int j = i+1; j < a.length; j++) {
            if( a[j].compareTo(a[smallestIndex]) < 0 ) {
                smallestIndex = j;
            }
        }
        E temp = a[i];
        a[i] = a[smallestIndex];
        a[smallestIndex] = temp;
    }
}
```

23

## Merge

Assuming left (L) and right (R) are sorted already, merge the two to create a new, single sorted array

L: 1 3 5 8      R: 2 4 6 7

How can we do this?

24

### Merge

L: 1 3 5 8      R: 2 4 6 7

Create a new array to hold the result that is the combined length

25

### Merge

L: 1 3 5 8      R: 2 4 6 7

What item is first?  
How did you know?

26

### Merge

L: 1 3 5 8      R: 2 4 6 7

Compare the first two elements in the lists!

27

### Merge

L: 1 3 5 8      R: 2 4 6 7

1

What item is second?  
How did you know?

28

### Merge

L: 1 **3** 5 8      R: **2** 4 6 7

1

Compare the **smallest element that hasn't been used yet** in each list

- For L, this is next element in the list
- For R, this is still the first element

29

### Merge

L: 1 **3** 5 8      R: **2** 4 6 7

1

General algorithm?

30

### Merge

L: 1 **3** 5 8      R: **2** 4 6 7

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

31

### Merge

L: 1 **3** 5 8      R: **2** 4 6 7

1

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

32



### Merge

L: 1 3 5 8      R: 2 4 6 7

1

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

33

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

34

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

35

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

36

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

37

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

38

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

39

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

40

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

41

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5 6

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

42

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5 6

General algorithm:

- Keep the index for where we are in each input array
- Start them both at 0
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that index

43

### Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5 6 7

What do we do now?

44

## Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5 6 7 8

If we run off the end of either array, just copy the remaining from the other array

45

## Merge in code

```
public static <E extends Comparable<E>> E[] merge(E[] left, E[] right) {
    int size = left.length+right.length;
    E[] result = (E[])new Object[size];

    int lIndex = 0;
    int rIndex = 0;

    for( int i = 0; i < size; i++ ) {
        if( lIndex >= left.length ) { // done with left
            result[i] = right[rIndex];
            rIndex++;
        } else if( rIndex >= right.length ) { // done with right
            result[i] = left[lIndex];
            lIndex++;
        } else if( left[lIndex].compareTo(right[rIndex]) <= 0 ) {
            result[i] = left[lIndex];
            lIndex++;
        } else {
            result[i] = right[rIndex];
            rIndex++;
        }
    }

    return result;
}
```

46

## MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

47

## MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

If the two halves are sorted, does MergeSort work?

48

## MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

What are we missing? Why does this work?

49

## MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

MergeSort is recursive. We're missing a base case!

50

## MergeSort: base case

7

Is this array sorted?

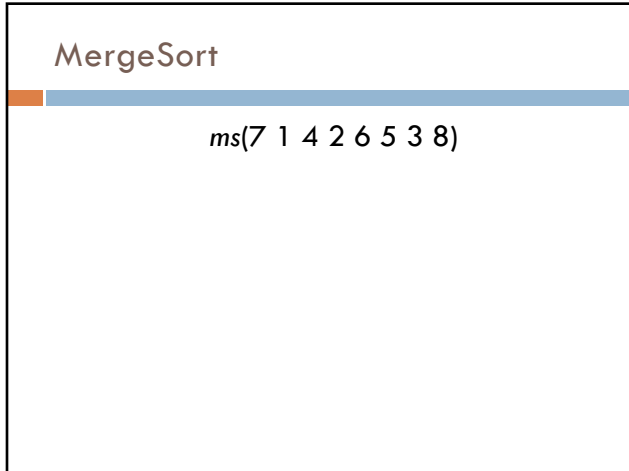
51

## MergeSort: base case

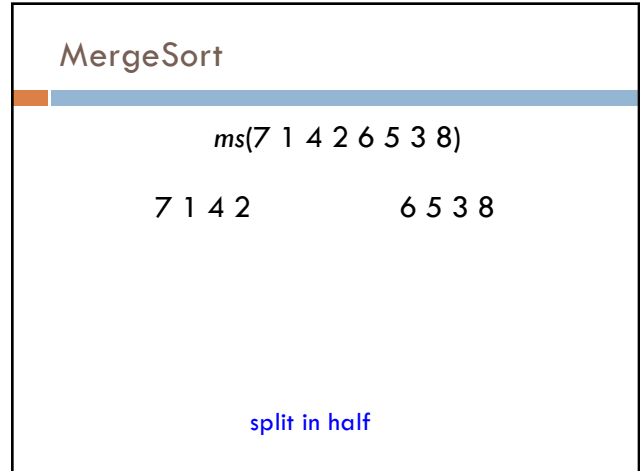
7

If the array is of size 1 (or 0), it's sorted

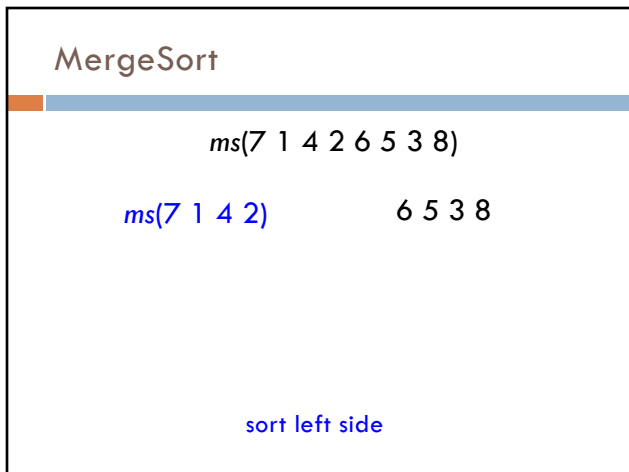
52



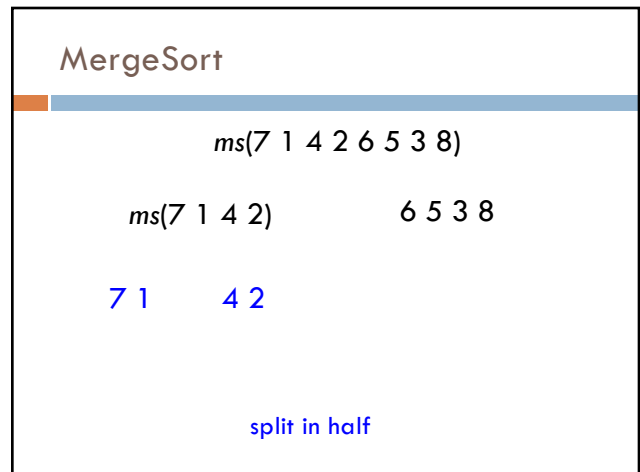
53



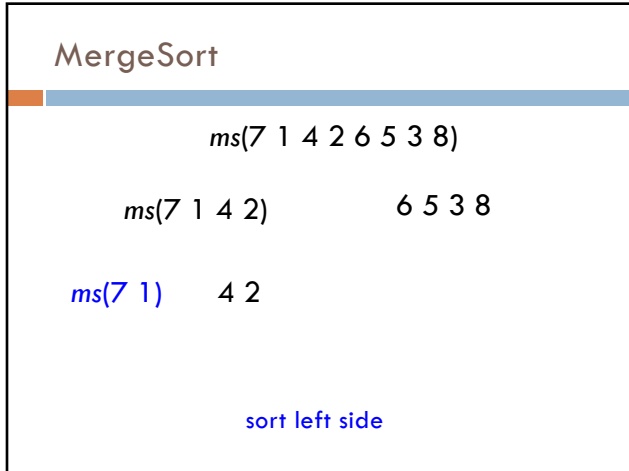
54



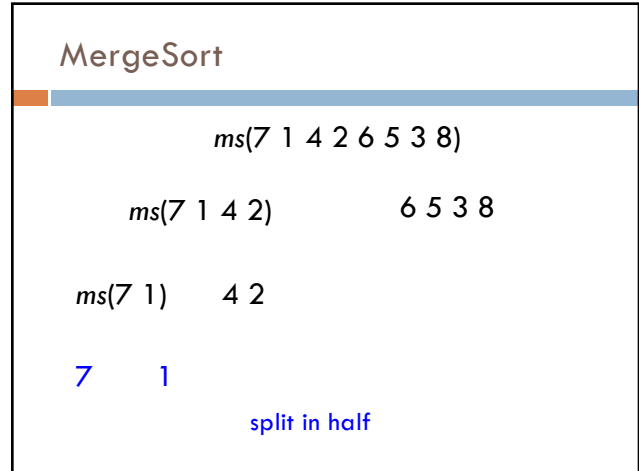
55



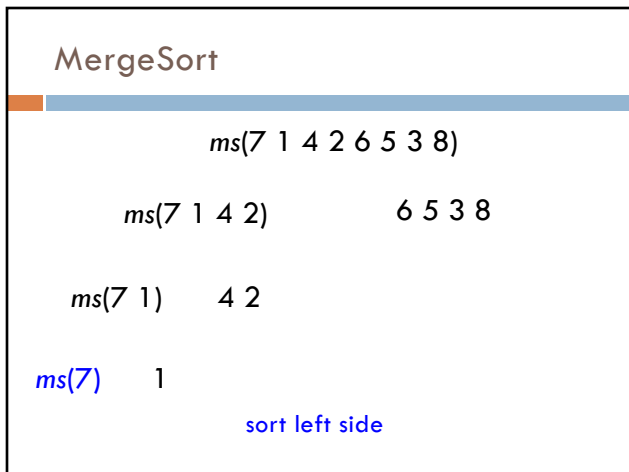
56



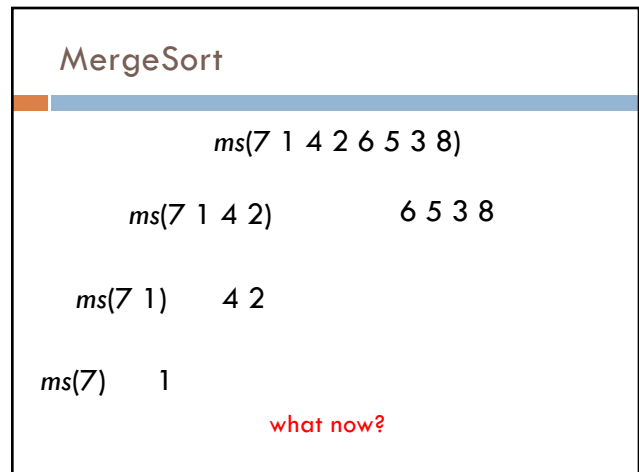
57



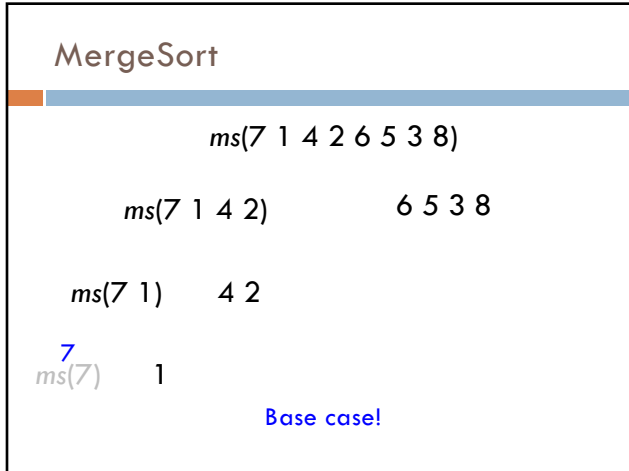
58



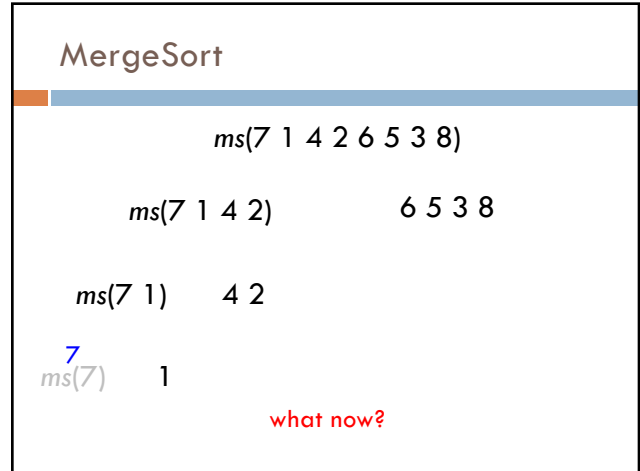
59



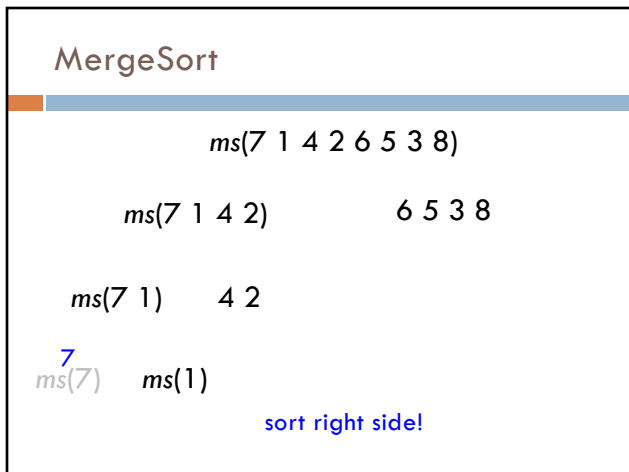
60



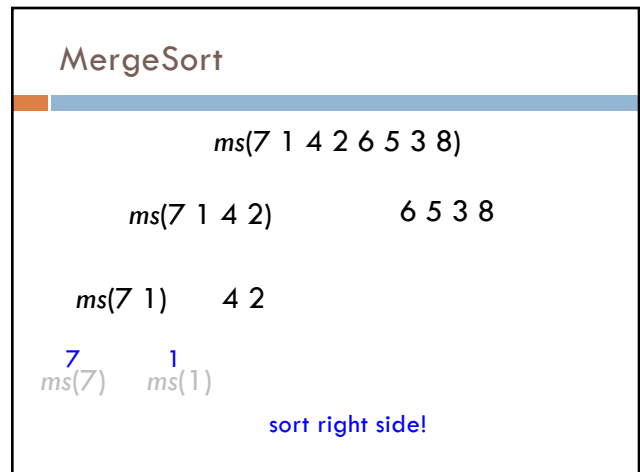
61



62

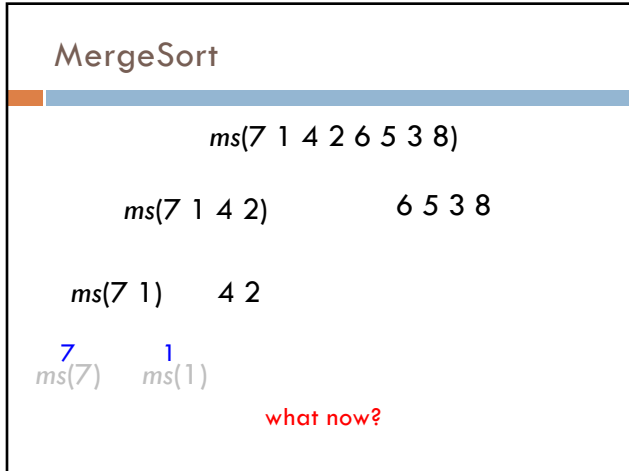


63

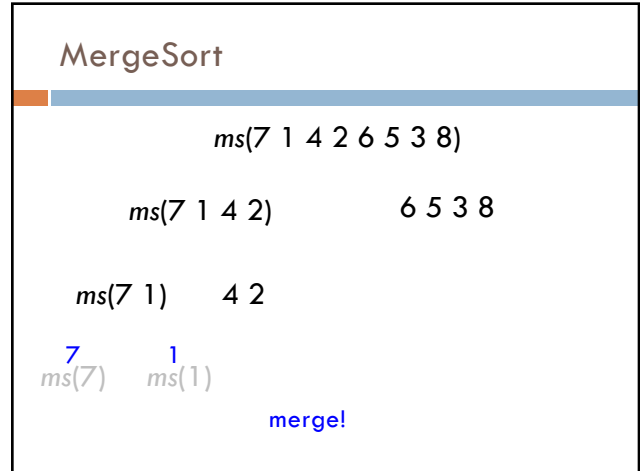


64

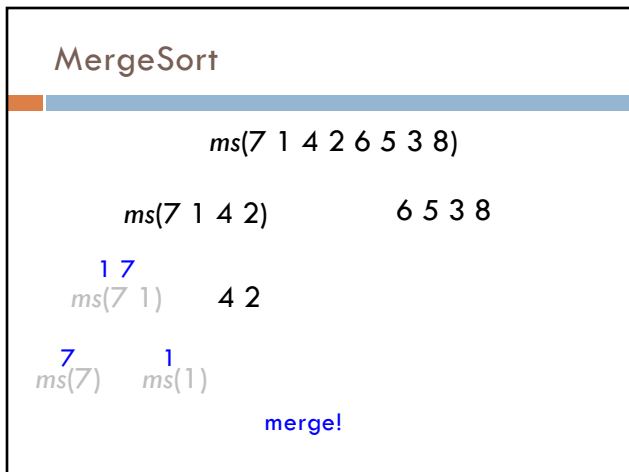




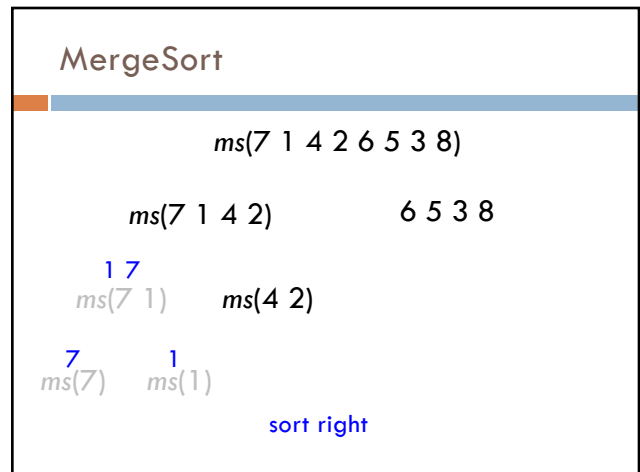
65



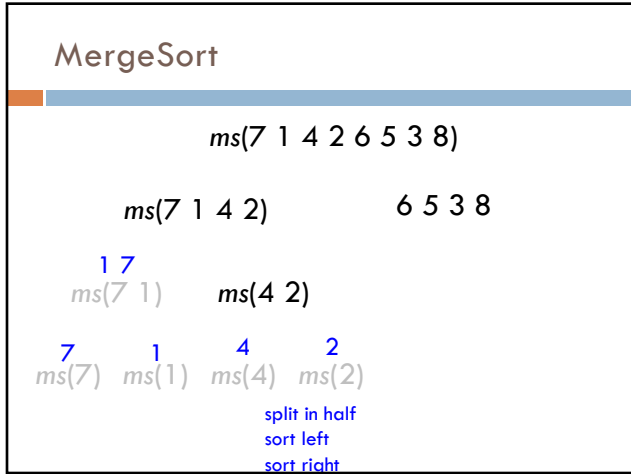
66



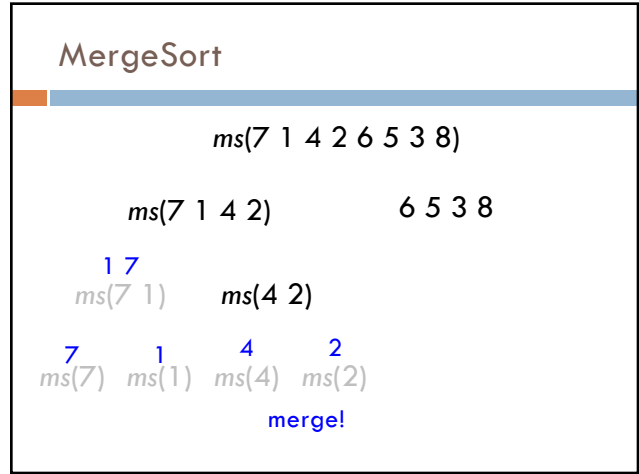
67



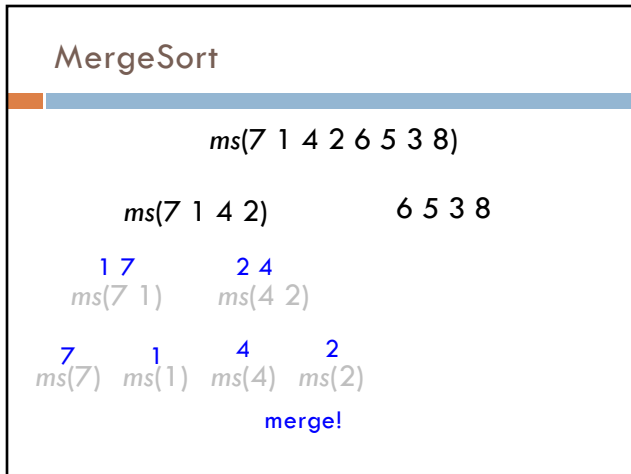
68



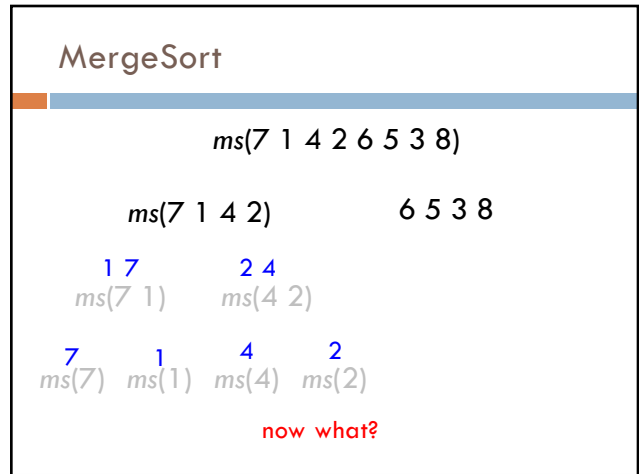
69



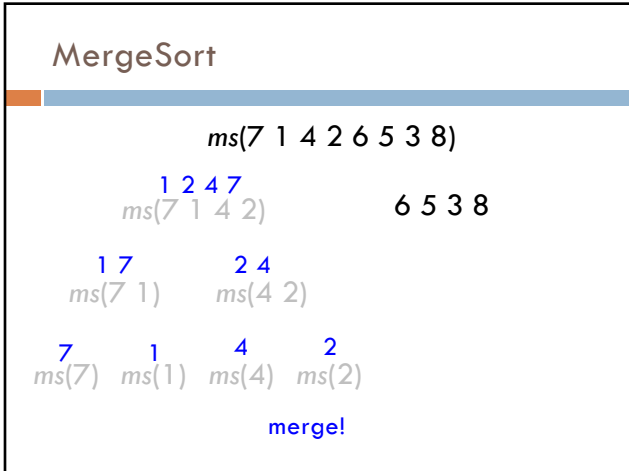
70



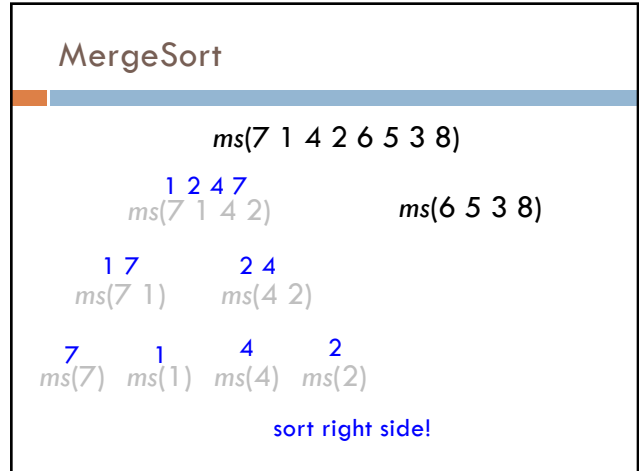
71



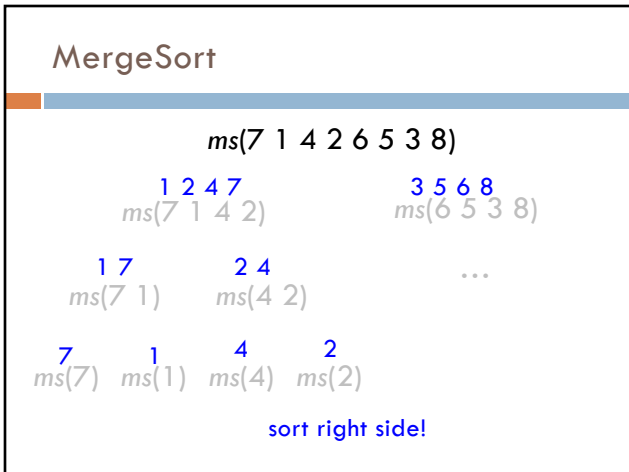
72



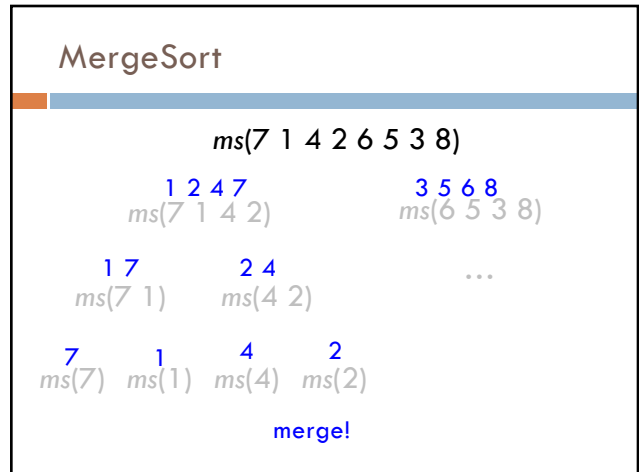
73



74



75



76

## MergeSort

```

1 2 3 4 5 6 7 8
ms(7 1 4 2 6 5 3 8)

1 2 4 7      3 5 6 8
ms(7 1 4 2)  ms(6 5 3 8)

1 7      2 4      ...
ms(7 1)  ms(4 2)

7      1      4      2
ms(7)  ms(1)  ms(4)  ms(2)

merge!

```

77

## MergeSort: implementation 1

```

public static <E extends Comparable<E>> E[] mergeSort(E[] a) {
    if ( a.length <= 1 ) {
        return a;
    } else {
        int mid = a.length/2;
        E[] left = Arrays.copyOfRange(a, 0, mid);
        E[] right = Arrays.copyOfRange(a, mid, a.length);
        E[] sortedLeft = mergeSort(left);
        E[] sortedRight = mergeSort(right);
        return merge(sortedLeft, sortedRight);
    }
}

```

78