

# MERGESORT

David Kauchak  
CS 62 – Fall 2026

1

## Admin

Assignment 5 (Wordle) out and due Friday (10/16)  
@ 5pm

Checkpoint 1 corrections

2

## 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)

3

## Selection sort

```
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;
    }
}
```

4

## 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;
    }
}
```

5

## Insertion sort

```
public static <E extends Comparable<E>> void insertSort(E[] a) {
    for( int i = 1; i < a.length; i++ ) {
        E current = a[i];
        int j = i-1;
        while( j >= 0 && current.compareTo(a[j]) < 0 ) {
            // move element j to element j+1
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = current;
    }
}
```

6

## 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?

7

## Merge

L: 1 3 5 8      R: 2 4 6 7

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

8

## Merge

L: 1 3 5 8      R: 2 4 6 7



What item is first?  
How did you know?

9

## Merge

L: 1 3 5 8      R: 2 4 6 7



Compare the first two elements in  
the lists!

10

## Merge

L: 1 3 5 8      R: 2 4 6 7



What item is second?  
How did you know?

11

## Merge

L: 1 3 5 8      R: 2 4 6 7



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

12

## Merge

L: 1 3 5 8      R: 2 4 6 7

1

General algorithm?

13

## 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

14

## 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

15

## 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

16

## 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

17

## 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

18

## 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

19

## 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

20

## 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

21

## 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

22

## 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

23

## 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

24

## 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

25

## 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

26

## Merge

L: 1 3 5 8      R: 2 4 6 7

1 2 3 4 5 6 7

What do we do now?

27

## 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

28

## 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 ) {
            result[i] = right[rIndex];
            rIndex++;
        } else if( rIndex >= right.length ) {
            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;
}
```

What are the different cases?

29

## 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;
}
```

30

## MergeSort

Divide the data in half

Recursively call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

31

## MergeSort

Divide the data in half

Recursively call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

If the two halves are sorted, does MergeSort work?

32

## MergeSort

Divide the data in half

Recursively call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

What are we missing? Why does this work?

33

## MergeSort

Divide the data in half

Recursively call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

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

34

## MergeSort: base case

7

Is this array sorted?

35

## MergeSort: base case

7

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

36

## MergeSort

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

37

## MergeSort

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

7 1 4 2

6 5 3 8

split in half

38

## MergeSort

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

6 5 3 8

sort left side

39

## MergeSort

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

6 5 3 8

7 1      4 2

split in half

40

## MergeSort

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

sort left side

41

## MergeSort

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

7      1

split in half

42

## MergeSort

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

sort left side

43

## MergeSort

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

what now?

44

## MergeSort

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

Base case!

45

## MergeSort

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

what now?

46

## MergeSort

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

sort right side!

47

## MergeSort

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

sort right side!

48

## MergeSort

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

what now?

49

## MergeSort

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

merge!

50

## MergeSort

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

merge!

51

## MergeSort

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

sort right

52

## MergeSort

*ms*(7 1 4 2 6 5 3 8)*ms*(7 1 4 2)      6 5 3 8*ms*(7 1)      *ms*(4 2)*ms*(7)   *ms*(1)   *ms*(4)   *ms*(2)split in half  
sort left  
sort right

53

## MergeSort

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

merge!

54

## MergeSort

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

merge!

55

## MergeSort

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

now what?

56

## MergeSort

$ms(7\ 1\ 4\ 2\ 6\ 5\ 3\ 8)$   
 $\begin{matrix} 1\ 2\ 4\ 7 \\ ms(7\ 1\ 4\ 2) \end{matrix} \quad 6\ 5\ 3\ 8$   
 $\begin{matrix} 1\ 7 & 2\ 4 \\ ms(7\ 1) & ms(4\ 2) \end{matrix}$   
 $\begin{matrix} 7 & 1 & 4 & 2 \\ ms(7) & ms(1) & ms(4) & ms(2) \end{matrix}$   
 merge!

57

## MergeSort

$ms(7\ 1\ 4\ 2\ 6\ 5\ 3\ 8)$   
 $\begin{matrix} 1\ 2\ 4\ 7 \\ ms(7\ 1\ 4\ 2) \end{matrix} \quad ms(6\ 5\ 3\ 8)$   
 $\begin{matrix} 1\ 7 & 2\ 4 \\ ms(7\ 1) & ms(4\ 2) \end{matrix}$   
 $\begin{matrix} 7 & 1 & 4 & 2 \\ ms(7) & ms(1) & ms(4) & ms(2) \end{matrix}$   
 sort right side!

58

## MergeSort

$ms(7\ 1\ 4\ 2\ 6\ 5\ 3\ 8)$   
 $\begin{matrix} 1\ 2\ 4\ 7 \\ ms(7\ 1\ 4\ 2) \end{matrix} \quad \begin{matrix} 3\ 5\ 6\ 8 \\ ms(6\ 5\ 3\ 8) \end{matrix}$   
 $\begin{matrix} 1\ 7 & 2\ 4 & & \\ ms(7\ 1) & ms(4\ 2) & & \dots \end{matrix}$   
 $\begin{matrix} 7 & 1 & 4 & 2 \\ ms(7) & ms(1) & ms(4) & ms(2) \end{matrix}$   
 sort right side!

59

## MergeSort

$ms(7\ 1\ 4\ 2\ 6\ 5\ 3\ 8)$   
 $\begin{matrix} 1\ 2\ 4\ 7 \\ ms(7\ 1\ 4\ 2) \end{matrix} \quad \begin{matrix} 3\ 5\ 6\ 8 \\ ms(6\ 5\ 3\ 8) \end{matrix}$   
 $\begin{matrix} 1\ 7 & 2\ 4 & & \\ ms(7\ 1) & ms(4\ 2) & & \dots \end{matrix}$   
 $\begin{matrix} 7 & 1 & 4 & 2 \\ ms(7) & ms(1) & ms(4) & ms(2) \end{matrix}$   
 merge!

60

## 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!

61

## 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);
    }
}
  
```

62

## 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);
    }
}
  
```

requires copying the data

63

## MergeSort: implementation 2

```

mergeSortHelper(data, low, high)
if high-low > 1
    midPoint = low + (high-low)/2

    mergeSortHelper(data, low, mid)
    mergeSortHelper(data, mid, high)

    merge(data, low, mid, high)
  
```

How is this different?

64

## MergeSort: implementation 2

```
mergeSortHelper(data, low, high)
  if high-low > 1
    midPoint = low + (high-low)/2

    mergeSortHelper(data, low, mid)
    mergeSortHelper(data, mid, high)

    merge(data, low, mid, high)
```

- doesn't require the extra copy!
- low/high specify the range we're sorting
- `merge = mergeSortHelper(data, 0, data.length)`

65

## Merge:

`merge(data, low, mid, high)`

Assume:

- data starting at low up to, but not including, mid is sorted
- data starting at mid up to, but not including, high is sorted

Goal:

- data from low up to, but not including, high is sorted

Note: merge still requires an extra helper array!

66

## MergeSort runtime

Divide the data in half

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

Merge the two halves

What is the runtime of mergesort??

67

## Merge runtime

```
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;
}
```

What is the runtime of merge?

68

## Merge runtime

```
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;
}
```

$O(\text{left.length} + \text{right.length}) = O(n)$

69

## MergeSort runtime

Divide the data in half

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

Merge the two halves

Ignoring the cost of the recursive call, how much work is done **per call** of MergeSort?

70

## MergeSort runtime

Divide the data in half

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

Merge the two halves

Ignoring the cost of the recursive call, how much work is done **per call** of MergeSort?  $O(n)$

71

## Mergesort runtime

$ms(\text{length}=n)$

work done  
 $n$

72

## Mergesort runtime

work done  $n$

$ms(length=n)$

$ms(n/2)$        $ms(n/2)$

73

## Mergesort runtime

work done  $n$

$ms(length=n)$

$ms(n/2)$        $ms(n/2)$

How much work is done for these calls?

74

## Mergesort runtime

work done  $n$

$ms(length=n)$

$ms(n/2)$        $ms(n/2)$        $n/2 + n/2 = n$

How much work is done for these calls?

75

## Mergesort runtime

work done  $n$

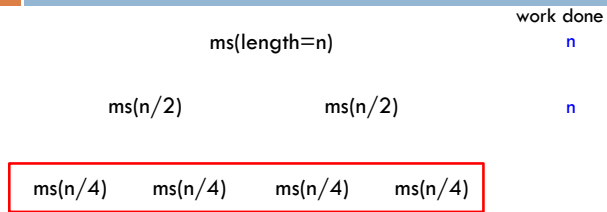
$ms(length=n)$

$ms(n/2)$        $ms(n/2)$        $n$

$ms(n/4)$        $ms(n/4)$        $ms(n/4)$        $ms(n/4)$

76

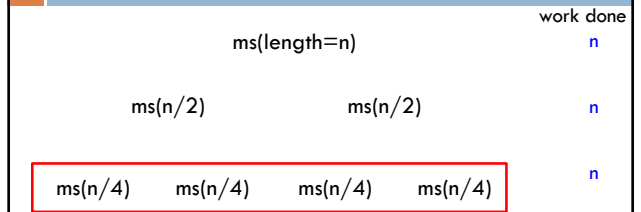
### Mergesort runtime



How much work is done for these calls?

77

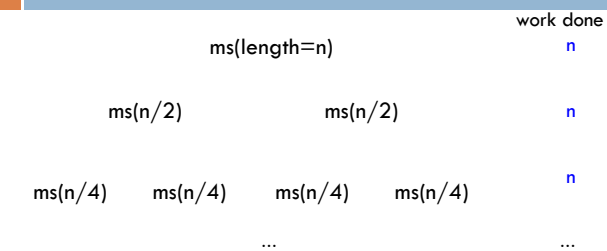
### Mergesort runtime



How much work is done for these calls?

78

### Mergesort runtime



How many levels are there?

79

### We've seen this before...

Each level down we decrease the size by 2:

$$length = \frac{n}{2^{level}}$$

80

## We've seen this before...

Each level down we decrease the size by 2:

$$\text{length} = \frac{n}{2^{\text{level}}}$$

We stop when the length is 1:

$$1 = \frac{n}{2^{\text{level}}}$$

$$2^{\text{level}} = n$$

$$\text{level} = \log_2 n$$

81

## MergeSort running time

$O(n)$  work at each level

$\log n$  levels

Overall runtime:  $O(n \log n)$  (best, worst, average)

82

## MergeSort properties

Stable?

In-place?

83

## MergeSort properties

Stable: yes!

```
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;
}
```

84

## MergeSort properties

Stable: yes

In-place: sort of

- easy way to implement is not
- very hard to do (but possible)
- in practice, variants of merge sort that combine with insertion sort are in-place