

# CSE 12 – Basic Data Structures and Object-Oriented Design

## Lecture 12

Greg Miranda & Paul Cao, Winter 2021

# Announcements

- Quiz 12 due Friday @ 8am
- Survey 5 due Friday @ 11:59pm
- PA4 due tonight @ 11:59pm
- PA5 released tomorrow
  - Closed for collaboration

# Topics

- Questions on Lecture 12?
- Combine/Sort

Questions on Lecture 12?

5 4 3 2 1

Bubble sort:

```
for index1 in 0 to n-1:
```

```
    for index2 in 0 to n-1:
```

```
        if (data[index2] > always true data[index2+1])
```

```
            swap(data[index2], data[index2+1])
```

$n-1$   $n-1$   
 $\bar{2}$   $\bar{2}$  4  
 $i=0$   $j=0$

) 4

$4n^2$

What is the worse case runtime of bubble sort?

A.  $O(n^2)$

B.  $O(n)$

C.  $O(n \log n)$

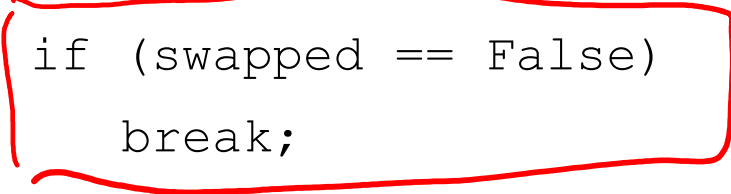
D. between  $O(n \log n)$  and  $O(n^2)$

E. Something else

4 3 5 2 1  
4 3 2 1 5

# Bubble Sort Optimization

```
for index1 in 0 to n-1:
    swapped = False
    for index2 in 0 to n-1-index1:
        if (data[index2] > data[index2+1])
            swap(data[index2], data[index2+1])
            swapped = True
    if (swapped == False)
        break;
```



Bubble sort optimized:

```
for index1 in 0 to n-1:
    swapped = False
    for index2 in 0 to n-1-index1:
        if (data[index2] > data[index2+1])
            swap(data[index2], data[index2+1])
            swapped = True
    if (swapped == False)
        break;
```

What is the worse case runtime of the optimized bubble sort?

- ☒ A.  $O(n^2)$
- ☐ B.  $O(n)$
- ☐ C.  $O(n \log n)$
- ☐ D. between  $O(n \log n)$  and  $O(n^2)$
- ☐ E. Something else

# Selection sort: Running time

## Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *find the position of the smallest element in the unsorted part*
  - *move this smallest element to the last position in the sorted part*
  - *increase the size of the sorted part and decrement the size of the unsorted part*

- Approximately how many times does the outer loop run?

A. 1 time

☒ B. N times

C.  $N^2$  times

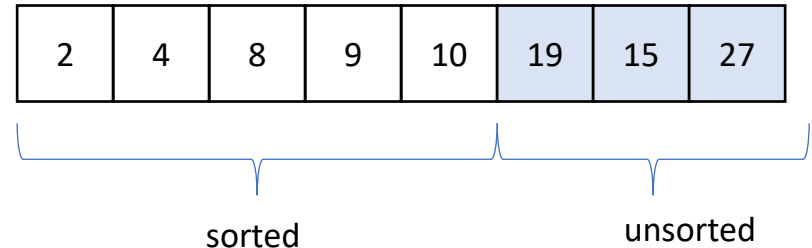




# Selection sort: Running time

## Pseudocode: selectionSort

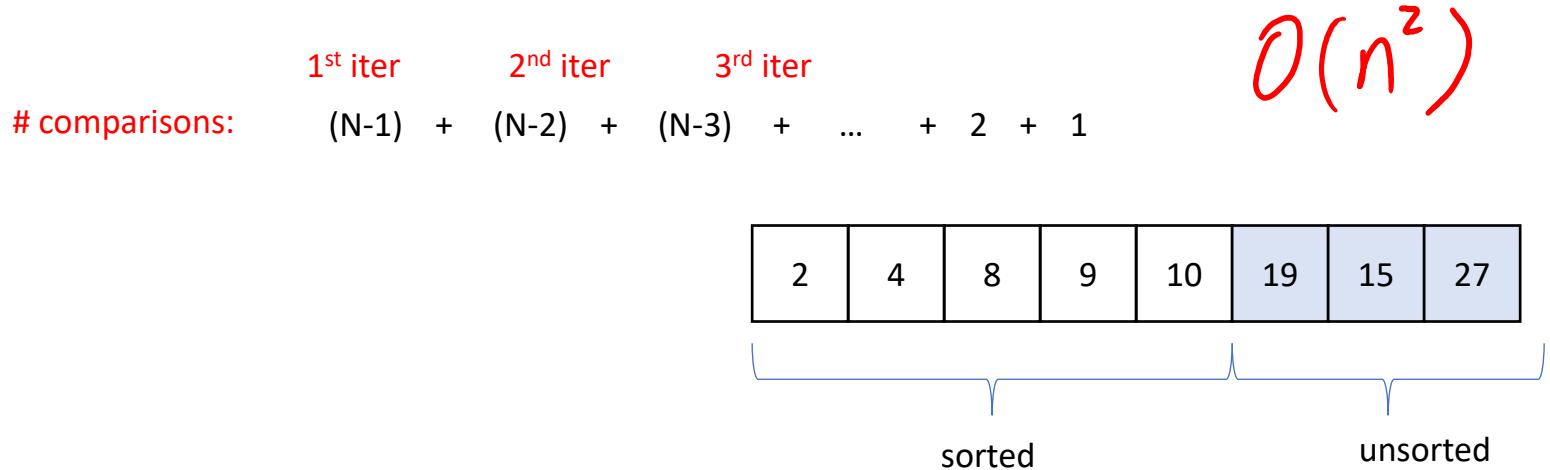
- *While the size of the unsorted part is greater than 1*
  - *find the position of the smallest element in the unsorted part*
  - *move this smallest element to the last position in the sorted part*
  - *increase the size of the sorted part and decrement the size of the unsorted part*
- Approximately how many comparisons does it take to find the smallest element in each iteration of the outer loop?
  - 1 comparison
  - Always  $N-1$  comparisons
  - At most  $N-1$ , but often less than  $N-1$



# Selection sort: Running time

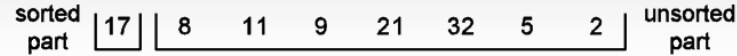
## Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *find the position of the smallest element in the unsorted part*
  - *move this smallest element to the last position in the sorted part*
  - *increase the size of the sorted part and decrement the size of the unsorted part*

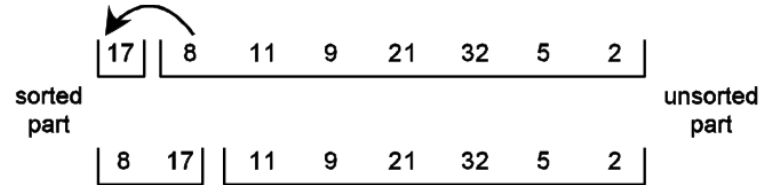


# Insertion Sort: The Picture

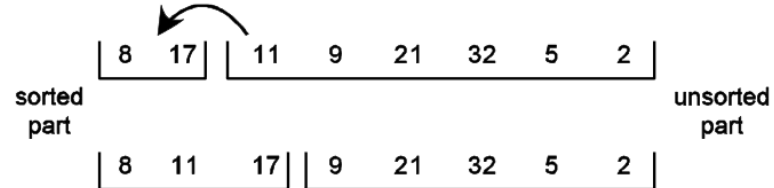
(a) Initial configuration for insertion sort. The input array is logically split into a sorted part (initially containing one element) and an unsorted part.



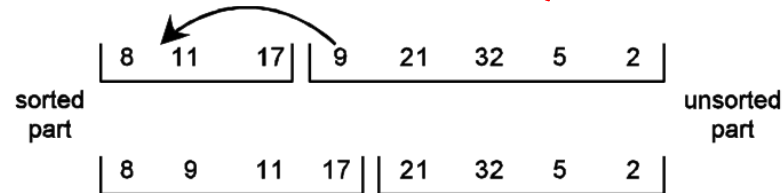
(b) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (first pass).



(c) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (second pass).



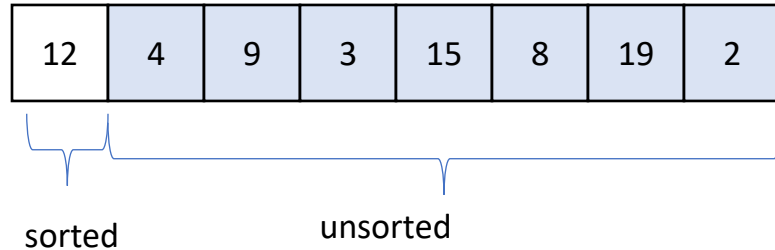
(d) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (third pass).



# Insertion sort: Worst case analysis

**# comparisons:**

	1 <sup>st</sup> iter	2 <sup>nd</sup> iter	3 <sup>rd</sup> iter	
	1	2	3	...
	+	+	+	...
				+
				(N-2)
				+
				(N-1)



# Insertion sort: Best case analysis

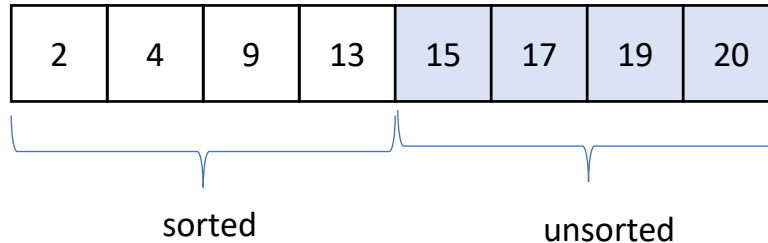
- Approximately how many steps does it take to insert the element into the sorted part each time through the loop in the BEST case?

A. 1

B.  $N$

C.  $N^2$

D. It depends on the length of the sorted part



# Array Sorting Algorithms (wiki)

Algorithm	Time Complexity		
	Best	Average	Worst
<a href="#">Quicksort</a>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
<a href="#">Mergesort</a>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
<a href="#">Timsort</a>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$
<a href="#">Heapsort</a>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
<a href="#">Bubble Sort</a>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
<a href="#">Insertion Sort</a>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
<a href="#">Selection Sort</a>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
<a href="#">Tree Sort</a>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
<a href="#">Shell Sort</a>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$
<a href="#">Bucket Sort</a>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$
<a href="#">Radix Sort</a>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
<a href="#">Counting Sort</a>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
<a href="#">Cubesort</a>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$

# MergeSort: The Magic of Recursion

[https://www.youtube.com/watch?v=XaqR3G\\_NVoo](https://www.youtube.com/watch?v=XaqR3G_NVoo)

- Consider this magical way of sorting lists:

12	4	9	3	15	8	19	2
----	---	---	---	----	---	----	---

Split the list in half:

12	4	9	3
----	---	---	---

15	8	19	2
----	---	----	---

Magically sort each list

3	4	9	12
---	---	---	----

2	8	15	19
---	---	----	----

Merge the two lists back together

2	3	4	8	9	12	15	19
---	---	---	---	---	----	----	----

# MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:

12	4	9	3	15	8	19	2
----	---	---	---	----	---	----	---

Split the list in half:

12	4	9	3
----	---	---	---

15	8	19	2
----	---	----	---

**Magically sort each list – using the same sorting method we are implementing!**

3	4	9	12
---	---	---	----

2	8	15	19
---	---	----	----

Merge the two lists back together



# MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:

12	4	9	3	15	8	19	2
----	---	---	---	----	---	----	---

Split the list in half:

12	4	9	3
----	---	---	---

15	8	19	2
----	---	----	---

**Magically sort each list – using the same sorting method we are implementing!**

3	4	9	12
---	---	---	----

2	8	15	19
---	---	----	----

Merge the two lists back together

```
public void mergeSort( int[] toSort )
{
    int mid = toSort.length / 2;
    int[] firstHalf = makeList(toSort, 0, mid);
    int[] secondHalf = makeList( toSort, mid, toSort.length );
    mergeSort( firstHalf );
    mergeSort( secondHalf );
    merge( firstHalf, secondHalf, toSort );
}
```

*Makes a new array and copies the  
elements from the specified range [a, b)*

*Recursion*

*Merges the first two lists together into the third,  
Maintaining sorted order*

Does this mergeSort method work?

A. Yes

☒ B. No

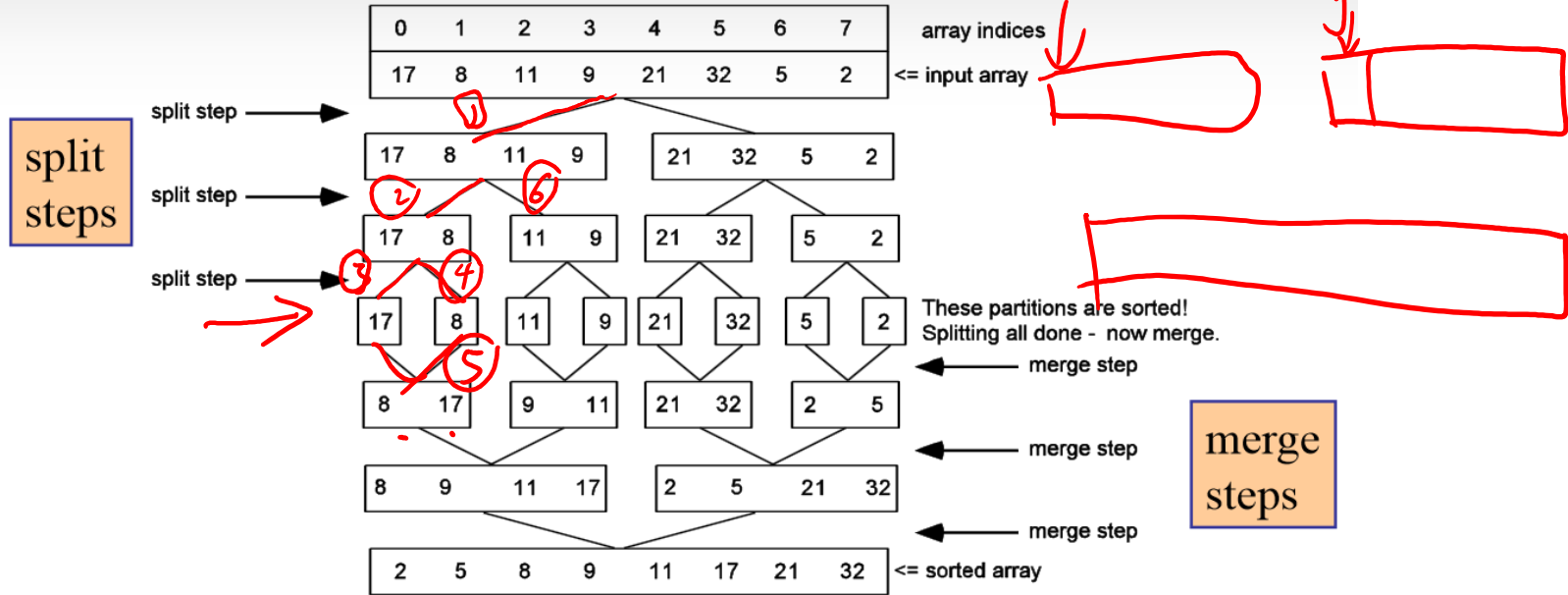
```
public void mergeSort( int[] toSort )
{
    if (toSort.length > 1) {
        int mid = toSort.length / 2;
        int[] firstHalf = makeList(toSort, 0, mid);
        int[] secondHalf = makeList( toSort, mid, toSort.length );
        mergeSort( firstHalf );
        mergeSort( secondHalf );
        merge( firstHalf, secondHalf, toSort );
    }
}
```

*Makes a new array and copies the  
elements from the specified range [a, b)*

*Merges the first two lists together into the third,  
Maintaining sorted order*

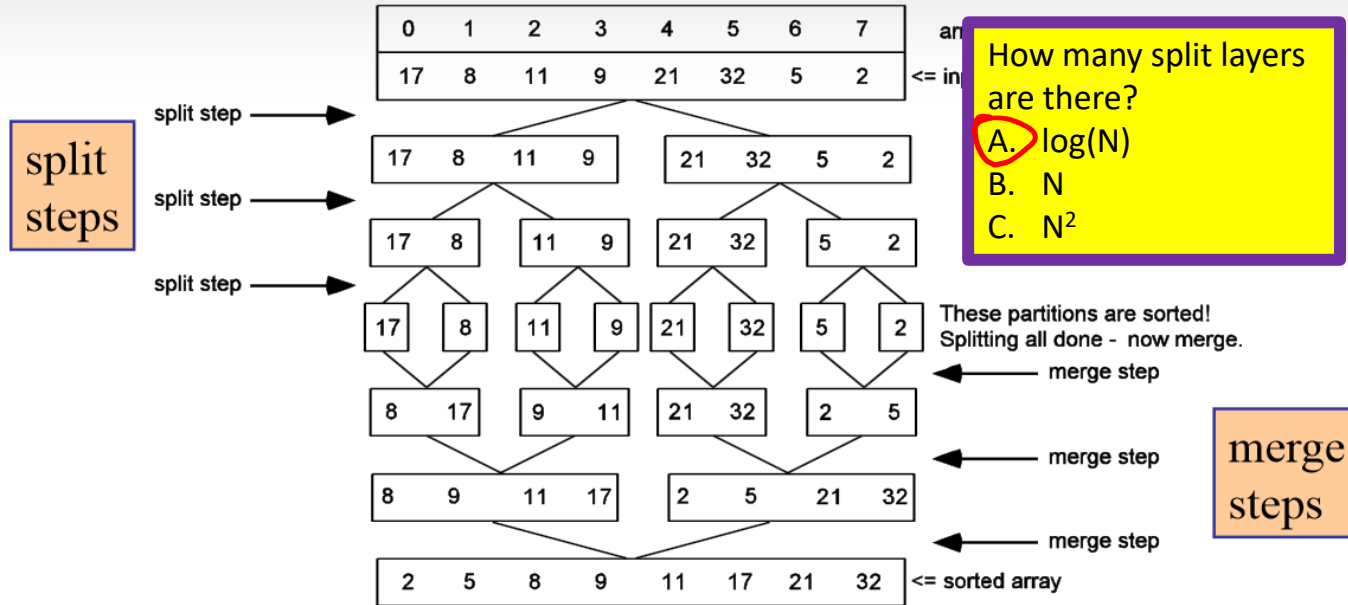
This mergeSort works!!

# Merge Sort: An Example



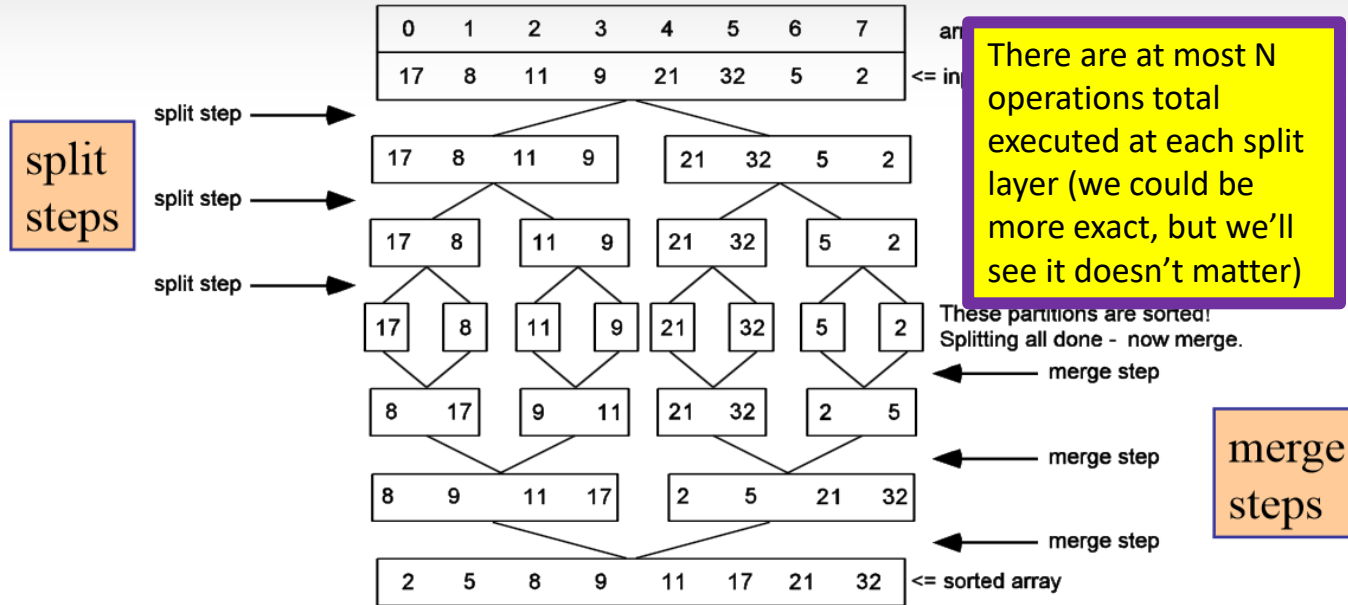
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions

# Merge Sort: An Example



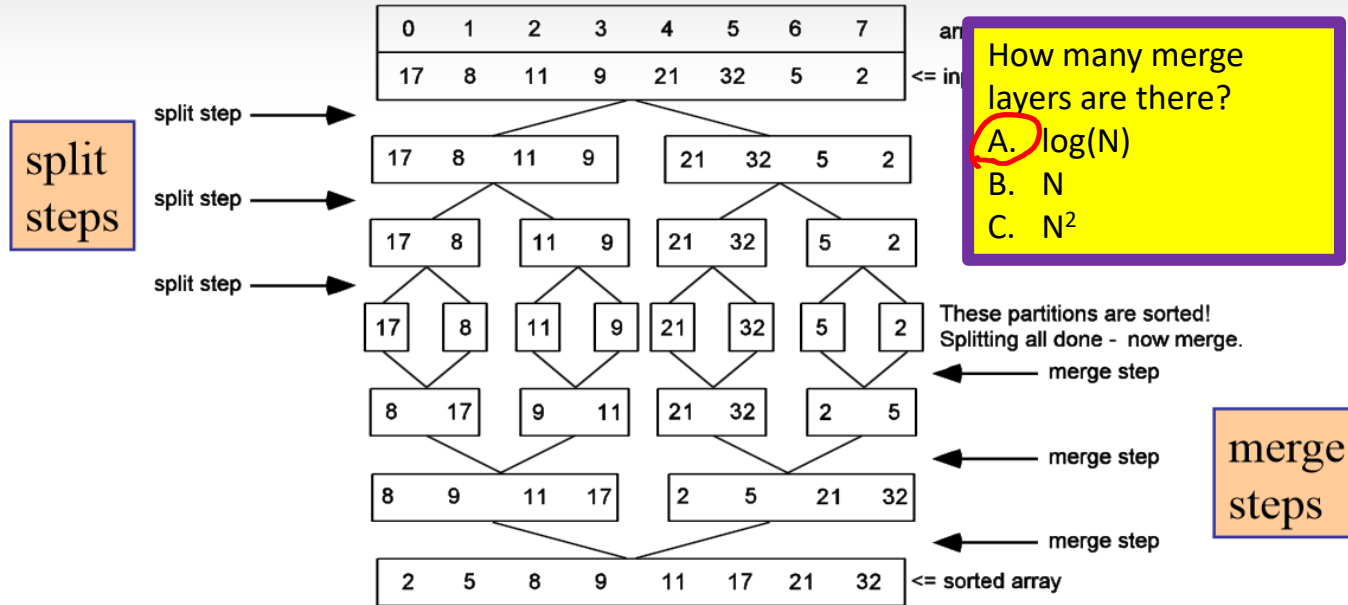
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions

# Merge Sort: An Example



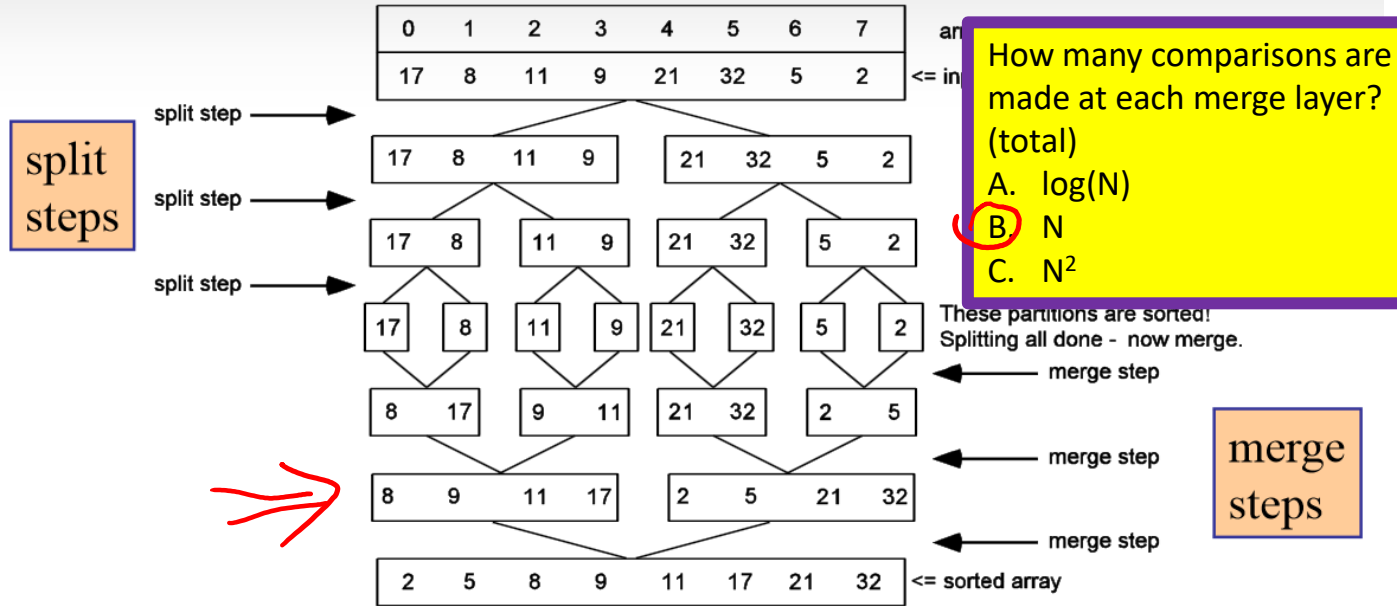
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions

# Merge Sort: An Example



Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions

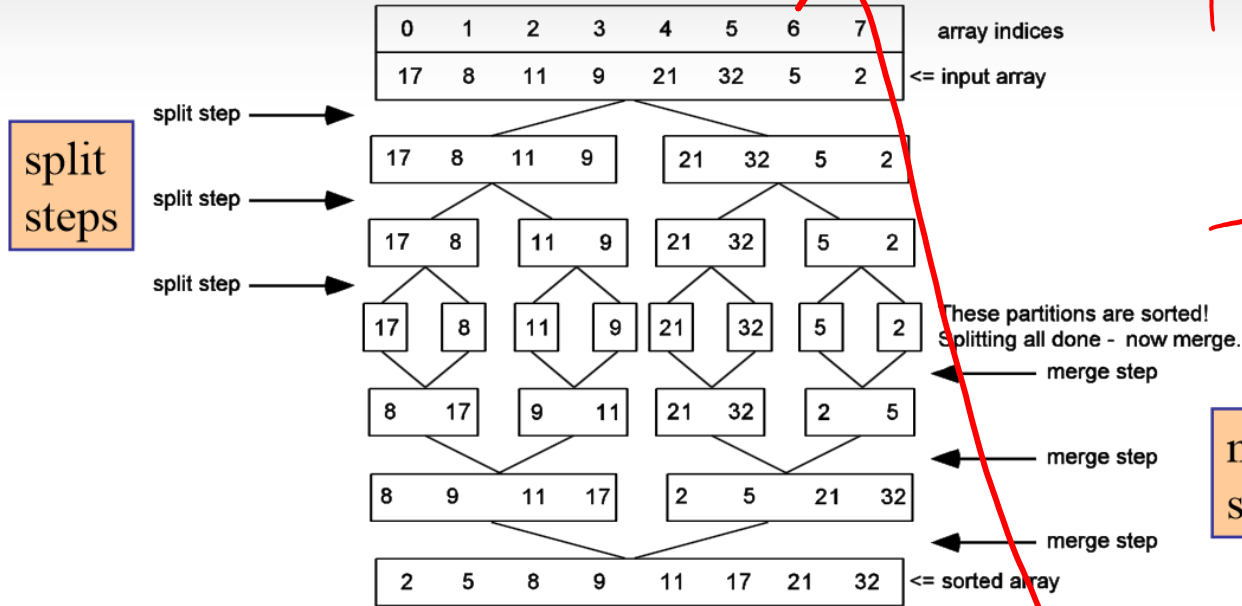
# Merge Sort: An Example



Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions



# Merge Sort: An Example



Running time for Mergesort:  
 $\log(N) \cdot N + \log(N) \cdot N = O(N \cdot \log(N))$

$$2 \log n T(n/2) + \log n \cdot n$$

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

$$T(n/2)$$

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

$$= 2(2T(n/4) + n/2)$$

$$= 4T(n/4) + n$$

$$= 8T(n/8) + 3n$$

```

public class SortFast {

    public static String s(int[] arr) { return Arrays.toString(arr); }

    public static int[] combine(int[] part1, int[] part2) {
        int index1 = 0, index2 = 0;
        int[] combined = new int[part1.length + part2.length];
        while(index1 < part1.length && index2 < part2.length) {
            if(part1[index1] < part2[index2]) {
                combined[index1 + index2] = part1[index1];
                index1 += 1;
            }
            else {
                combined[index1 + index2] = part2[index2];
                index2 += 1;
            }
        }
        while(index1 < part1.length) {
            combined[index1 + index2] = part1[index1]; index1 += 1;
        }
        while(index2 < part2.length) {
            combined[index1 + index2] = part2[index2]; index2 += 1;
        }
        System.out.println(s(part1) + " + " + s(part2) + " -> " + s(combined));
        return combined;
    }
}

```

```

public static int[] sortC(int[] arr) {

    if(arr.length <= 1) { return arr; }

    else {

        int[] part1 = Arrays.copyOfRange(arr, 0, arr.length / 2);

        int[] part2 = Arrays.copyOfRange(arr, arr.length / 2, arr.length);

        System.out.println(s(arr) + " -> " + s(part1) + " + " + s(part2));

        int[] sortedPart1 = sortC(part1);

        int[] sortedPart2 = sortC(part2);

        int[] sorted = combine(sortedPart1, sortedPart2);

        return sorted;

    }
}

public static void main(String[] args) {

    int[] result = SortFast.sortC(new int[]{34, 93, 12, 49, 69, 25, 39 });

    System.out.println(SortFast.s(result));
}

```