CSE 12 – Basic Data Structures: Sorting

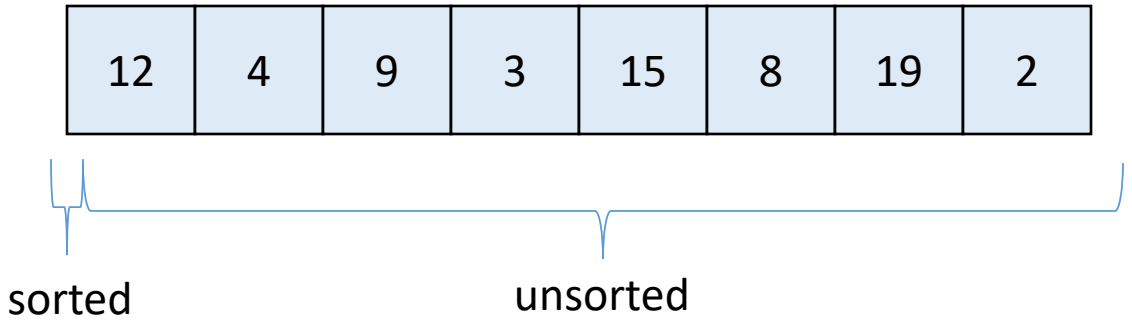
Prof. Joe Politz

Slides adapted from Prof. Christine Alvarado

Practical sorting algorithms: Selection sort

Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *Find the position of the smallest element in the unsorted part*
  - *Swap this smallest element with the first position in the unsorted part*
  - *Increase the size of the sorted part and decrement the size of the unsorted part*

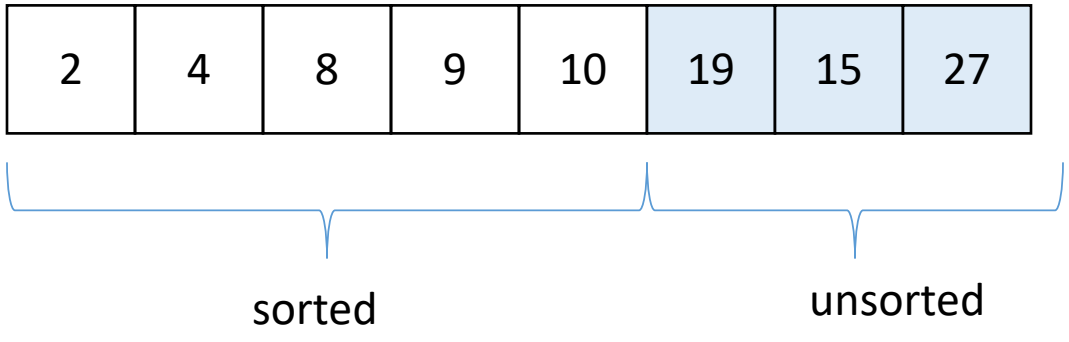


Selection sort: Running time

Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *Find the position of the smallest element in the unsorted part*
  - *Swap this smallest element with the first position in the unsorted 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

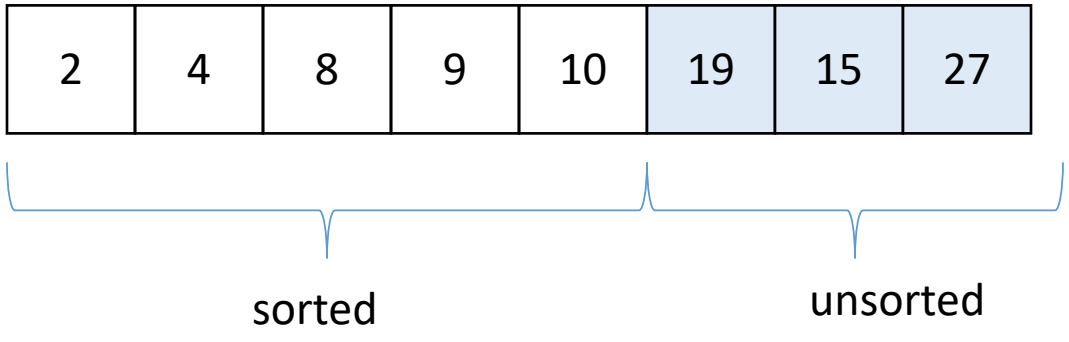


Selection sort: Running time

Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *Find the position of the smallest element in the unsorted part*
  - *Swap this smallest element with the first position in the unsorted 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?
  A. 1 comparison
  B. Always N-1 comparisons
  C. At most N-1, but often less than N-1



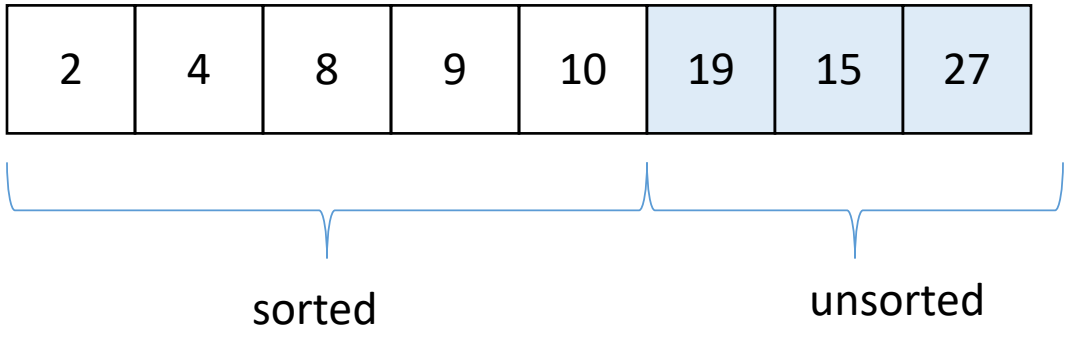
Selection sort: Running time

Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *Find the position of the smallest element in the unsorted part*
  - *Swap this smallest element with the first position in the unsorted part*
  - *Increase the size of the sorted part and decrement the size of the unsorted part*

1st iter 2nd iter 3rd iter

comparisons: (N-1) + (N-2) + (N-3) + … + 2 + 1



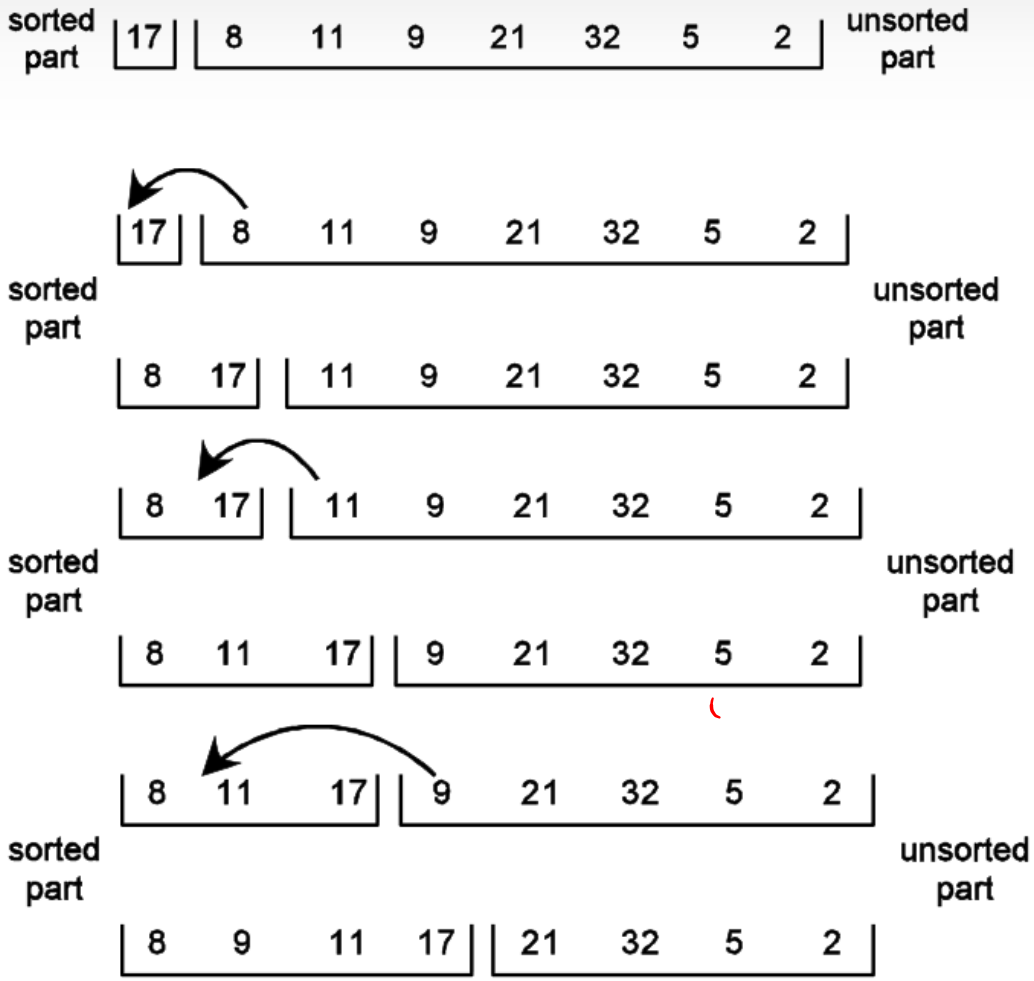
Insertion Sort: The Picture

(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).

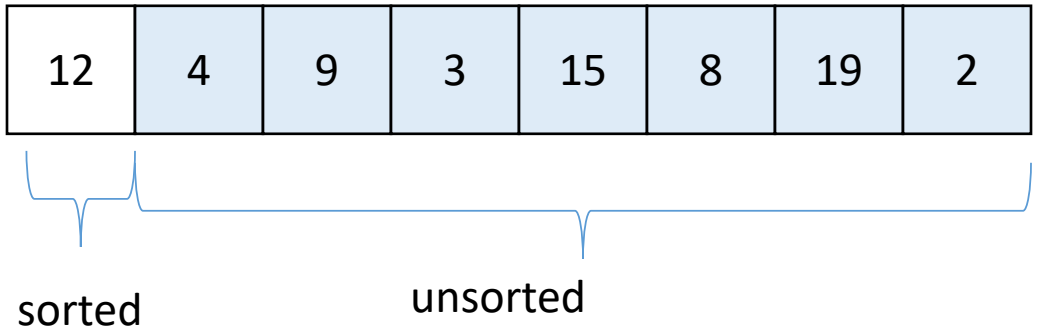
(d) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (third pass).



Insertion sort: Worst case analysis

1st iter 2nd iter 3rd iter

comparisons: 1 + 2 + 3 + … + (N-2) + (N-1)



Insertion sort: Best case analysis

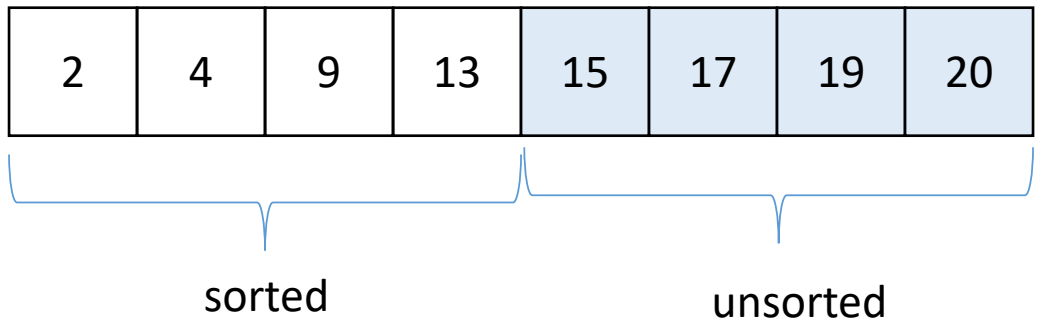
- Approximately how many steps does it take to insert the element into the sorted part each time through the loop in the BEST case?

A. 1

B. N

C. N^2

D. It depends on the length of the sorted part



Insertion sort: Best case analysis

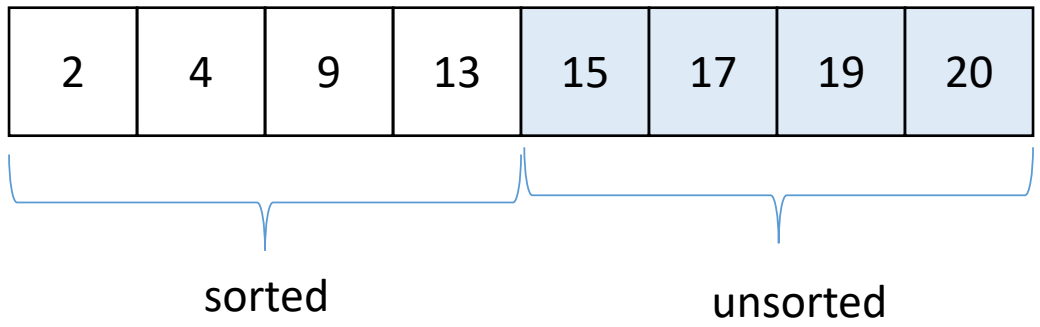
- Approximately how many steps does it take to insert the element into the sorted part each time through the loop in the WORST case?

A. 1

B. N

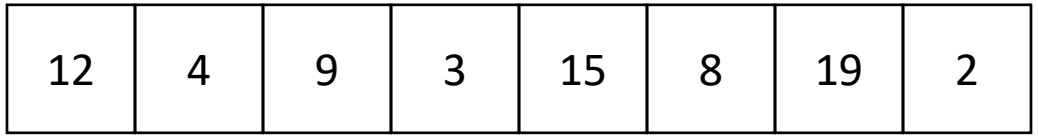
C. N^2

D. It depends on the length of the sorted part

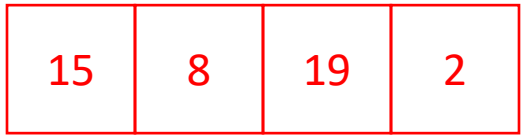


MergeSort: The Magic of Recursion

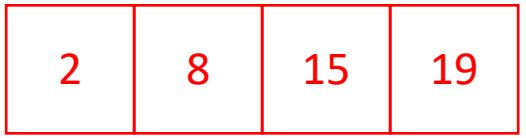
- Consider this magical way of sorting lists:



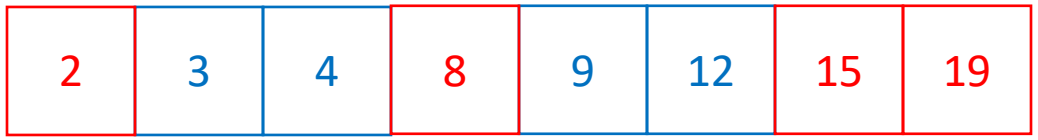
Split the list in half:



Magically sort each list

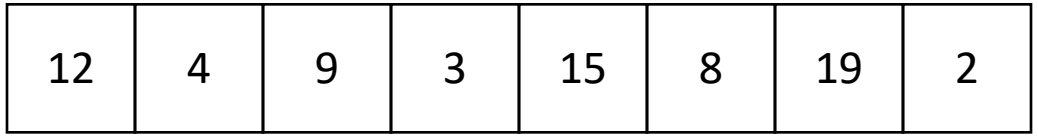


Merge the two lists back together

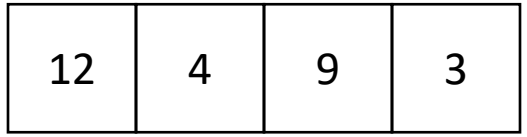


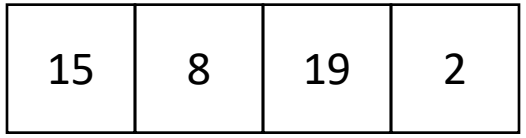
MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:

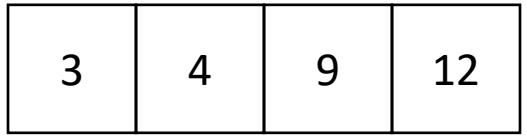


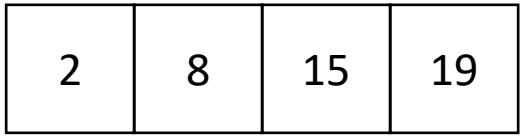
Split the list in half:





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

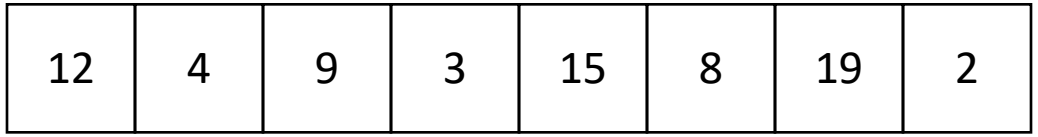




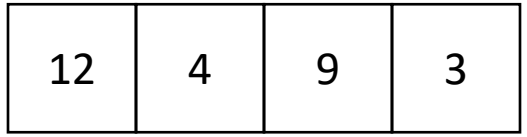
Merge the two lists back together

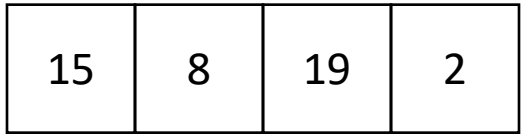
MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:

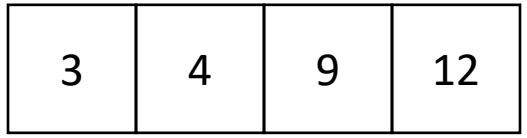


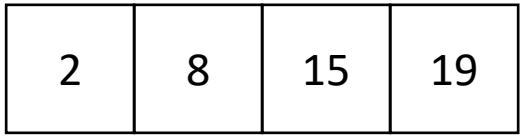
Split the list in half:





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





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

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

Does this mergeSort method work?

A. Yes
B. No

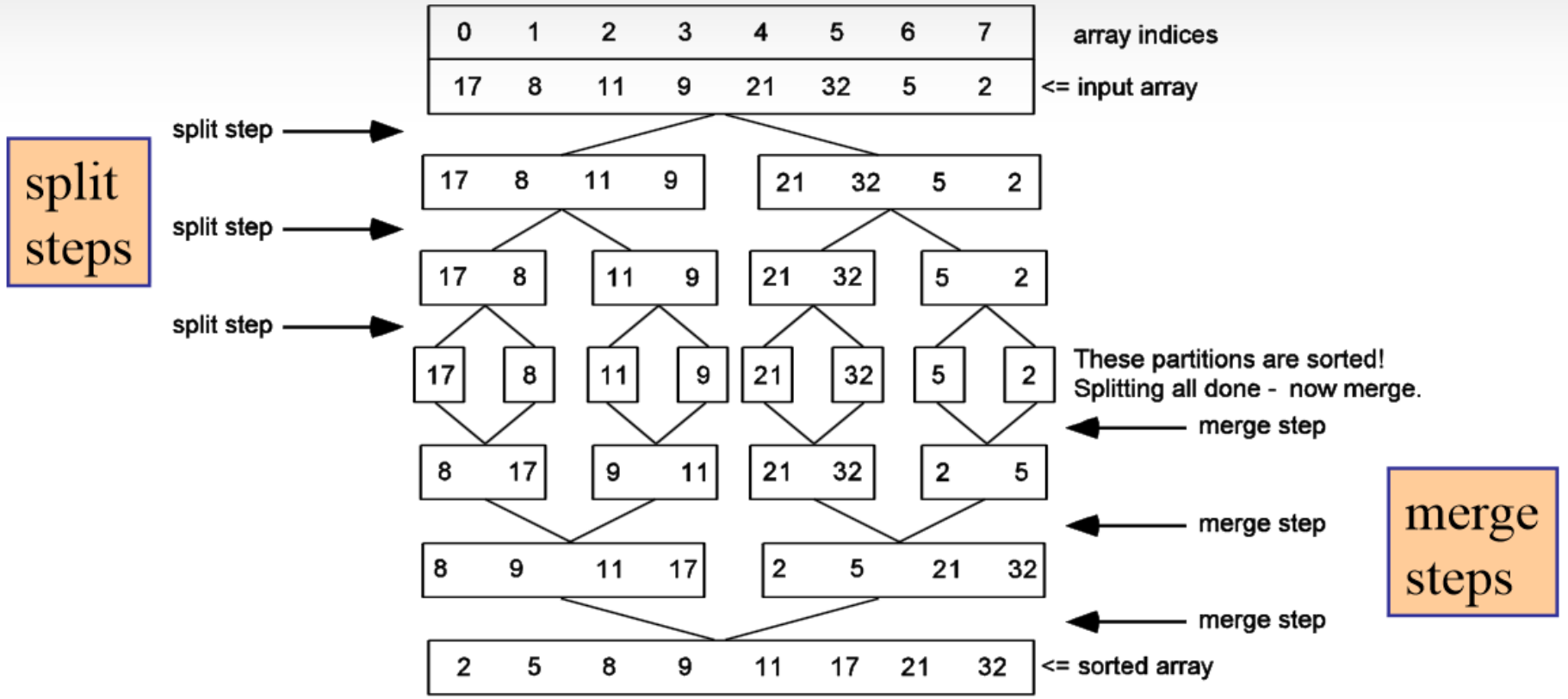
```
public void mergeSort( int[] toSort )
{
  if (toSort.length > 1) {
    int mid = toSort.length / 2;
    int[] firstHalf = makeList(toSort, 0, mid);
    int[] secondHalf = makeList( toSort, mid, toSort.length );
    mergeSort( firstHalf );
    mergeSort( secondHalf );
    merge( firstHalf, secondHalf,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

Makes a new array and copies the elements from the specified range

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

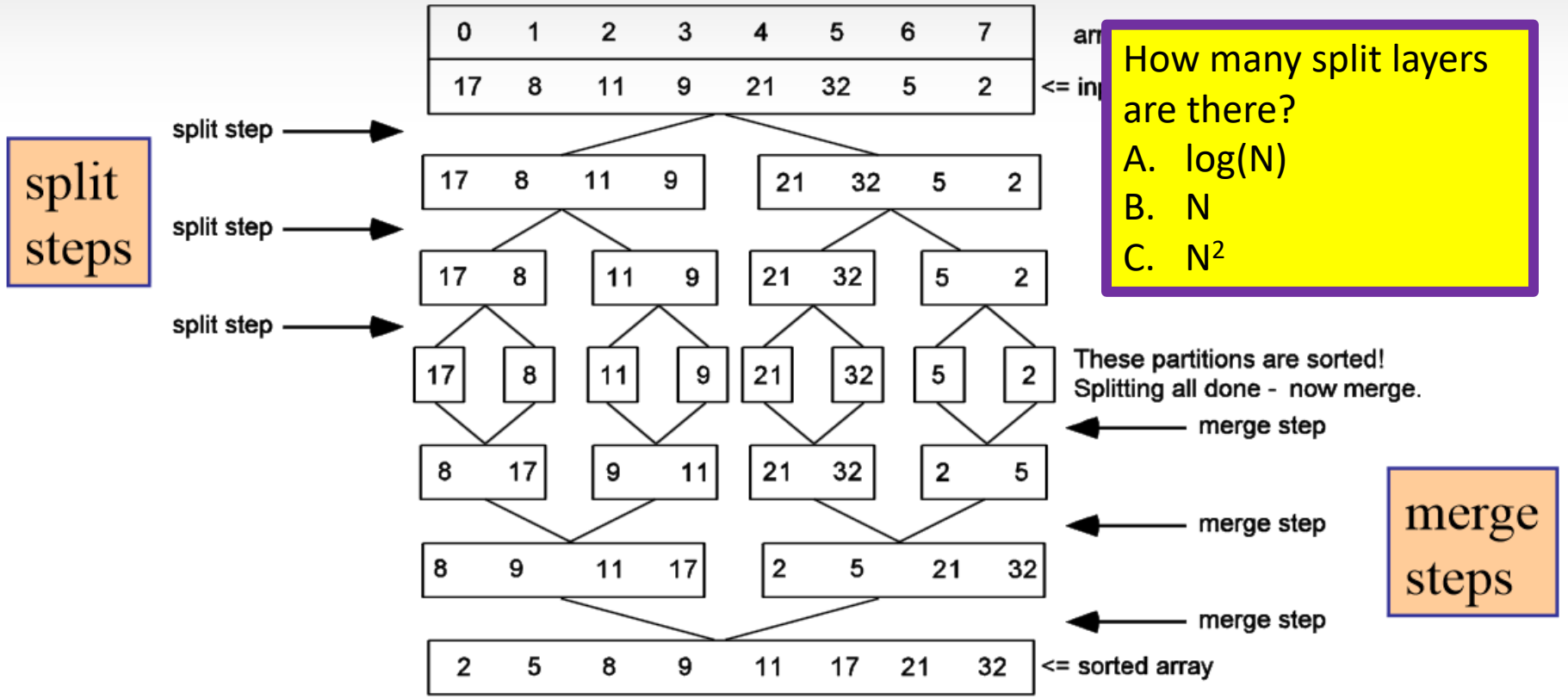
This mergeSort works!!

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

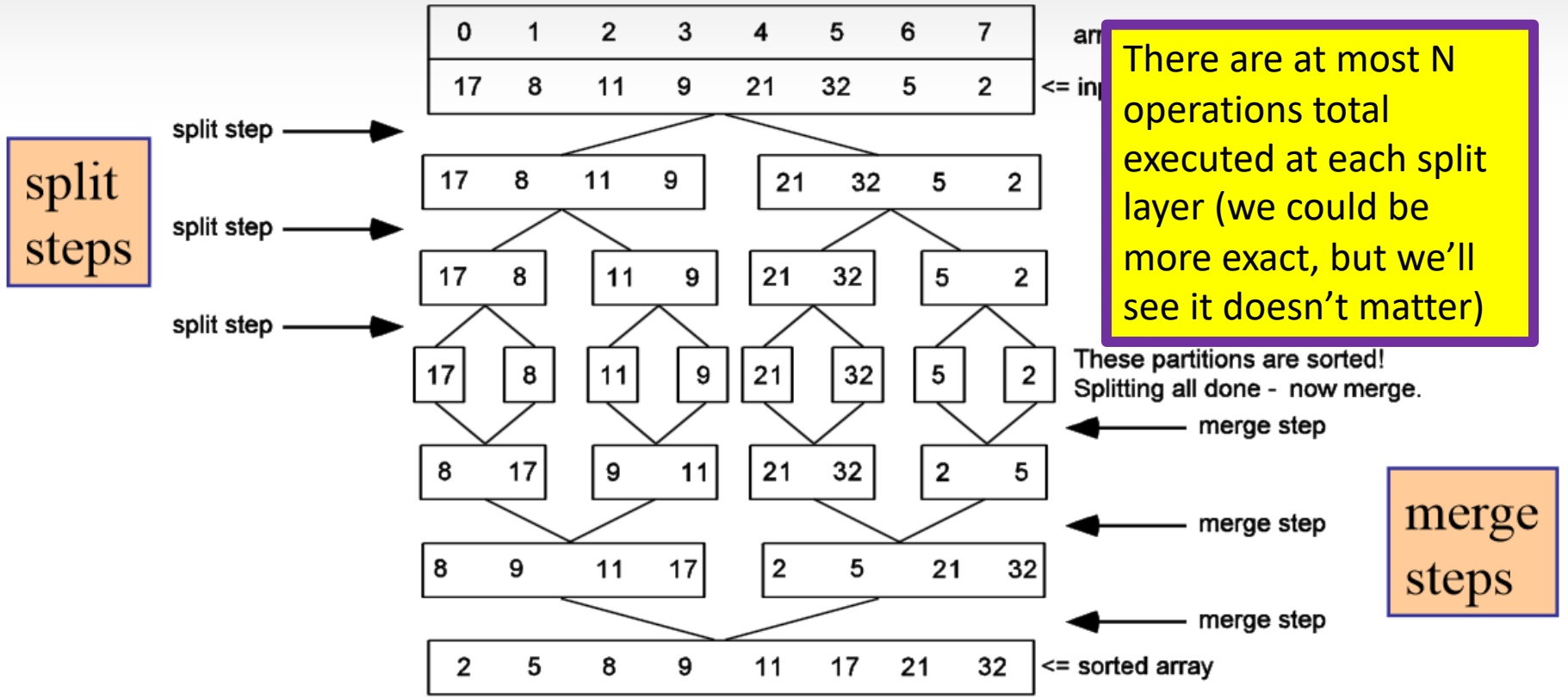


How many split layers are there?

A. log(N)
B. N
C. N^2

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

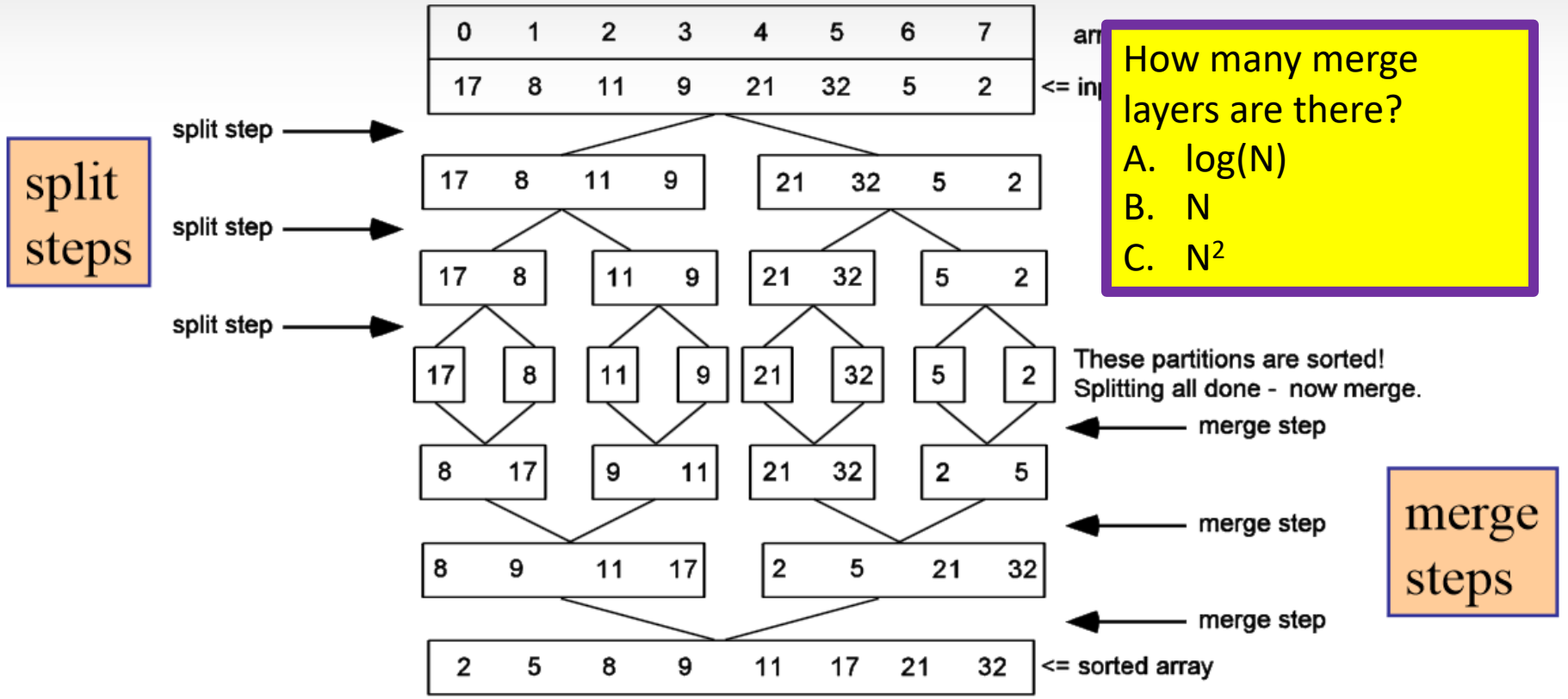
Merge Sort: An Example



There are at most N operations total executed at each split layer (we could be more exact, but we'll see it doesn't matter)

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

Merge Sort: An Example

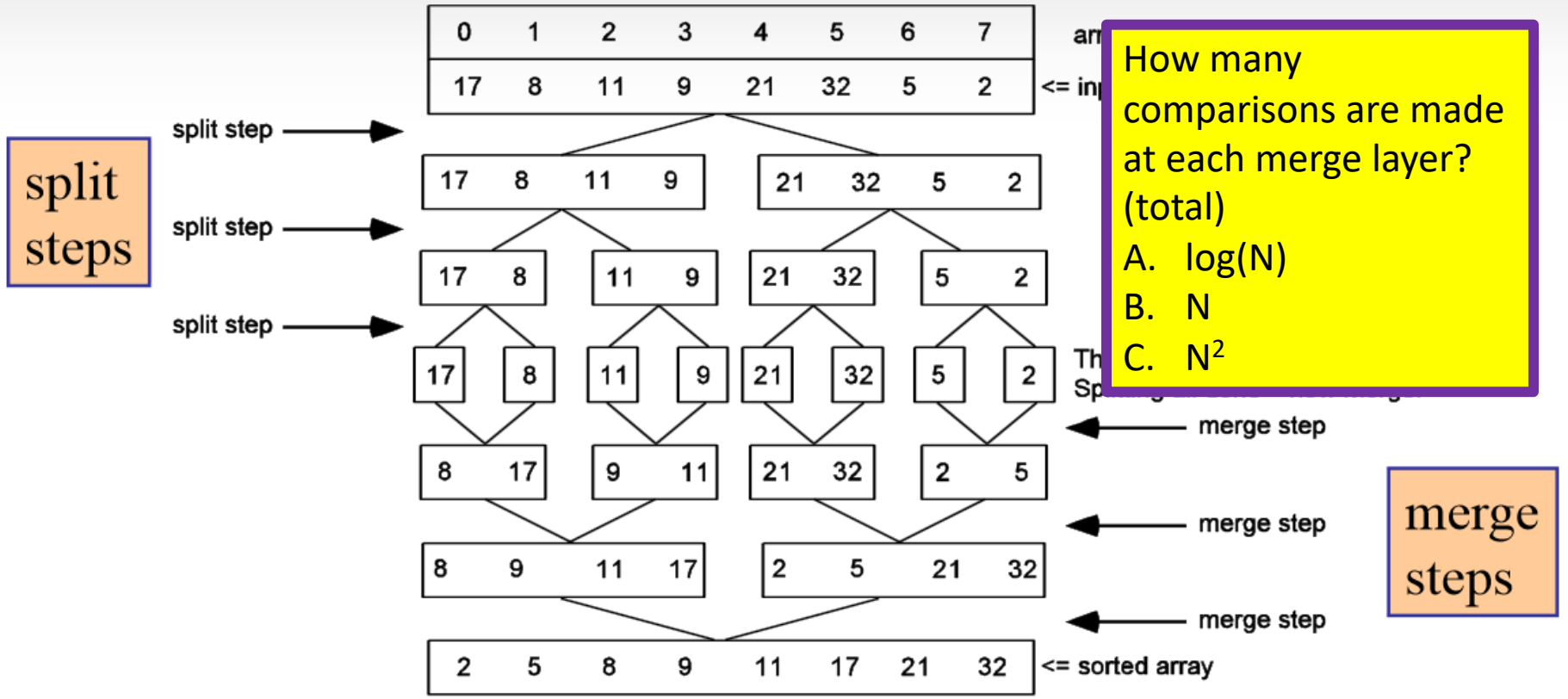


How many merge layers are there?

A. log(N)
B. N
C. N^2

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

Merge Sort: An Example



split steps

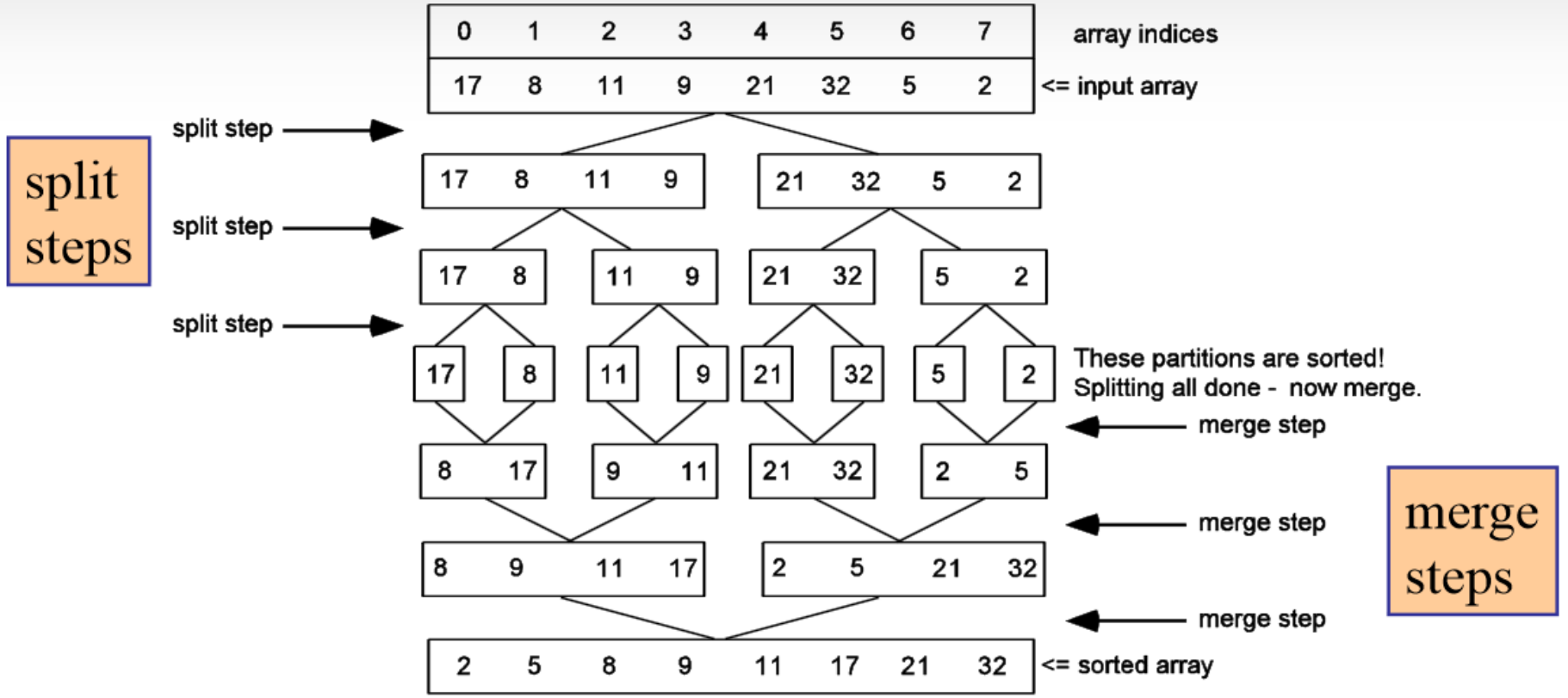
merge steps

How many comparisons are made at each merge layer? (total)

A. log(N)
B. N
C. N^2

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)*N + log(N)*N = O(N*log(N))

Quicksort: Another magical (recursive) algorithm

14	4	9	12	15	8	19	2

Select a **pivot** element:

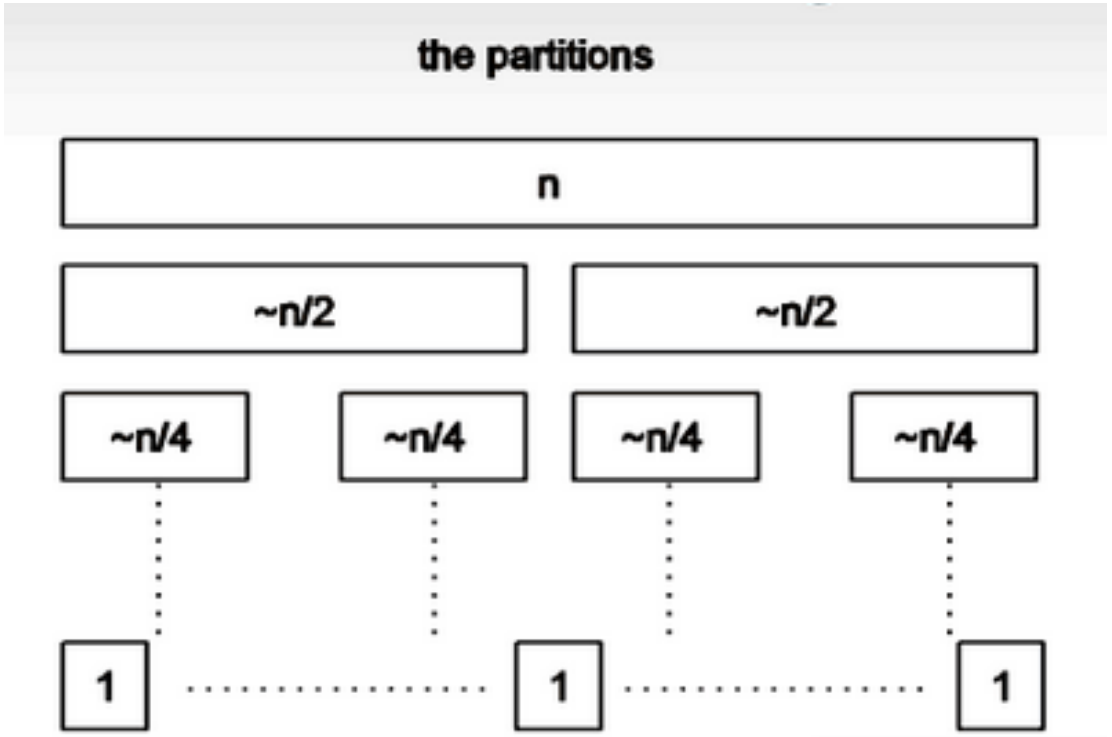
14	4	9	**12**	15	8	19	2

“Partition” the elements in the array (smaller or equal to pivot, larger or equal to pivot)

2	4	9	8	15	**12**	19	14

The book and your next HW describe how this partition step works. You'll work with it on your HW.

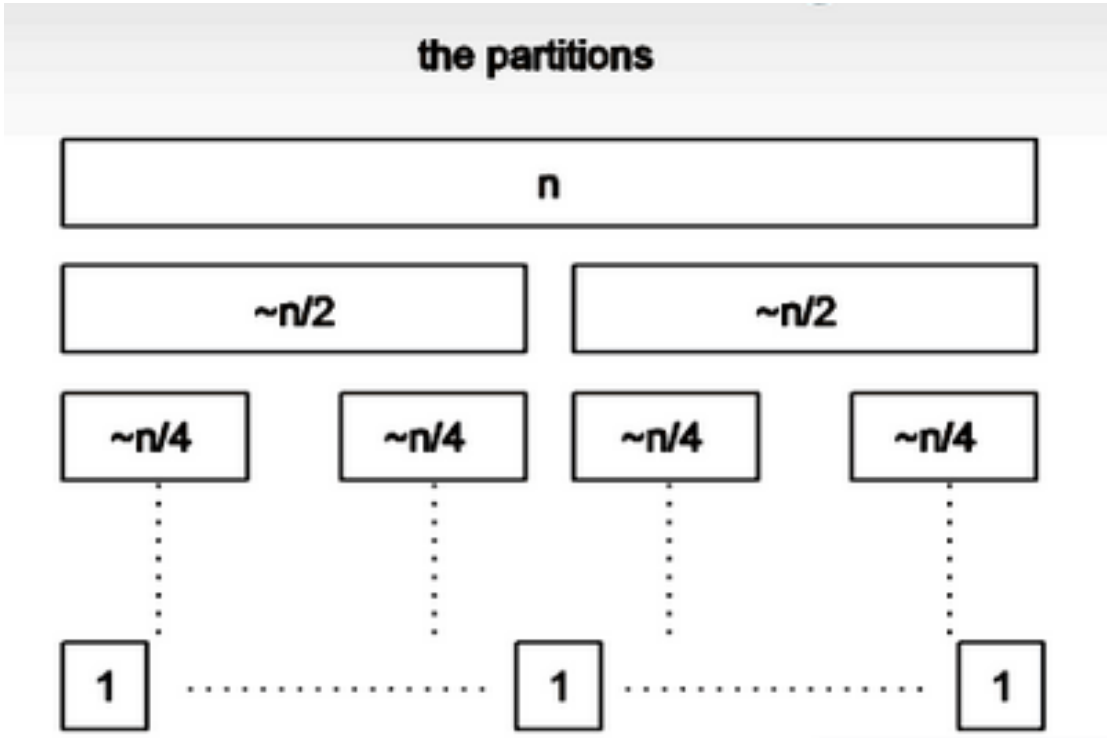
Quick Sort: Using a “good” pivot



How many levels will there be if you choose a pivot that divides the list in half?

A. 1
B. log(N)
C. N
D. N*log(N)
E. N^2

Quick Sort: Using a “good” pivot



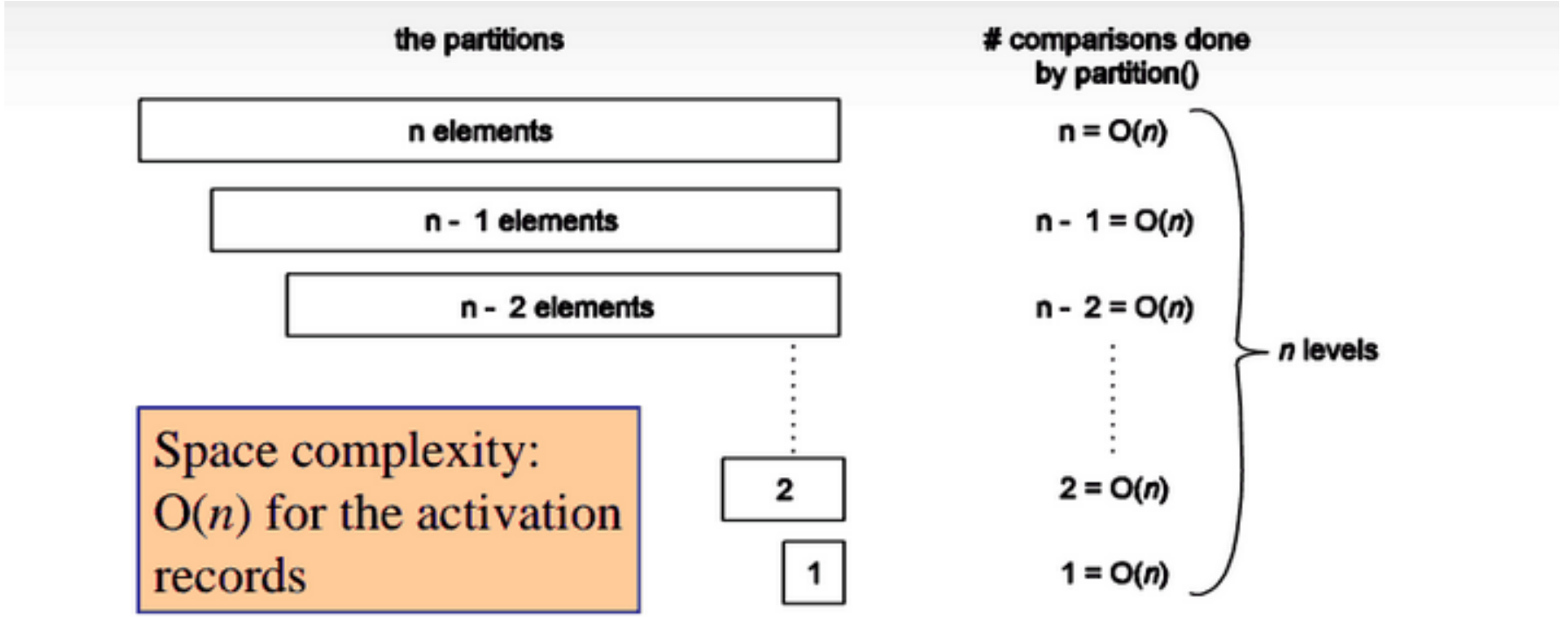
If the time to partition on each level takes N comparisons, how long does Quicksort take with a good partition?

A. O(1)
B. O(log(N))
C. O(N)
D. O(N*log(N))
E. $O(N^2)$

Which of these choices would be the *worst* choice for the pivot?

A. The minimum element in the list
B. The last element in the list
C. The first element in the list
D. A random element in the list

Quick sort with a bad pivot



Space complexity: O(n) for the activation records

If the pivot always produces one empty partition and one with $n - 1$ elements, there will be n levels, each of which requires O(n) comparisons: O(n^2) time complexity

Which of these choices is a better choice for the pivot?

A. The first element in the list
B. A random element in the list
C. They are about the same

Next time

- More sorting and starting with C!