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

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

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?

```
Bubble sort:
for index1 in 0 to n-1:
  for index2 in 0 to n-1:
    if (data[index2] > data[index2+1])
      swap(data[index2], data[index2+1])
```

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

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?

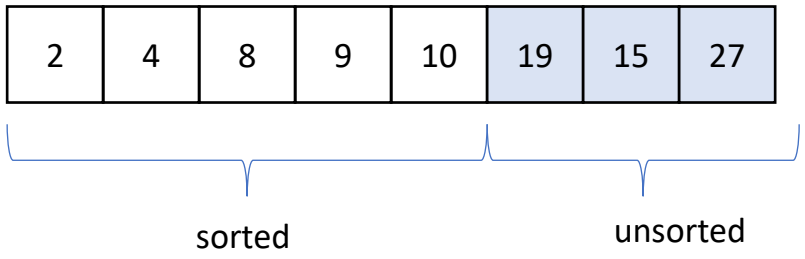
A. $O(n^2)$
B. O(n)
C. O(nlogn)
D. between O(nlogn) and $O(n^2)$
E. Something else

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

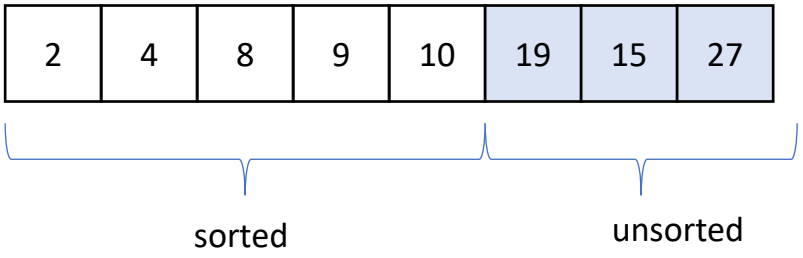


Selection sort: Running time

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?
  A. 1 comparison
  B. Always N-1 comparisons
  C. At most N-1, but often less than N-1



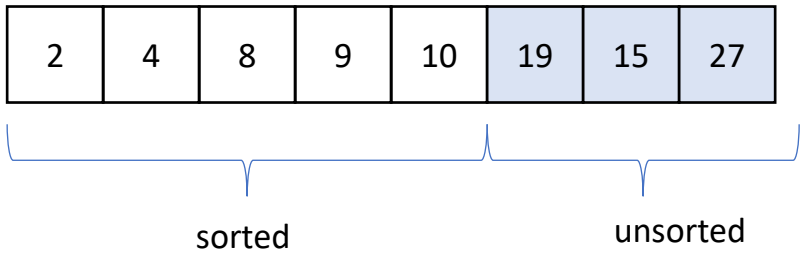
Selection sort: Running time

Pseudocode: selectionSort

- *While the size of the unsorted part is greater than 1*
  - *find the position of the smallest element in the unsorted part*
  - *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*

1st iter 2nd iter 3rd iter

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



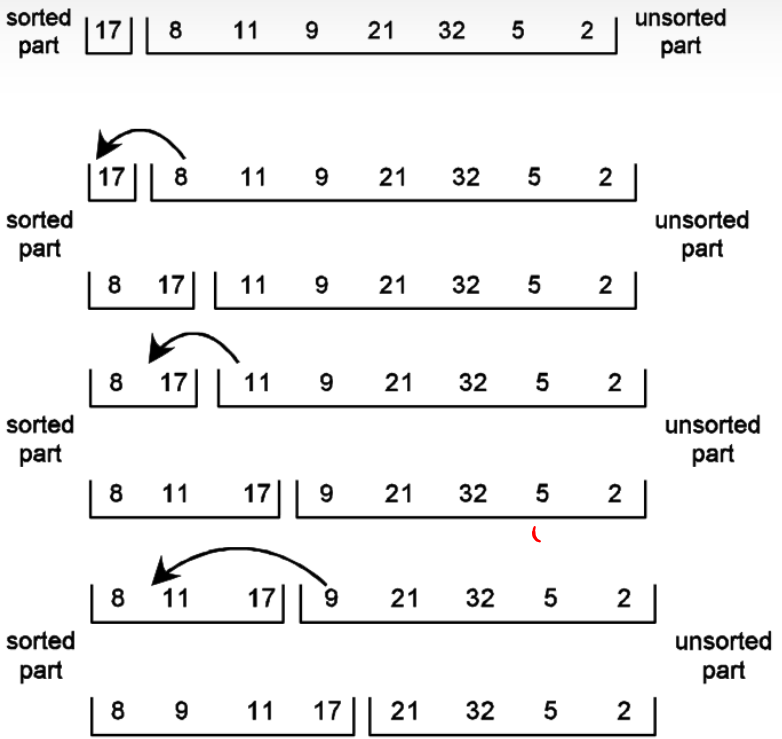
Insertion Sort: The Picture

(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).

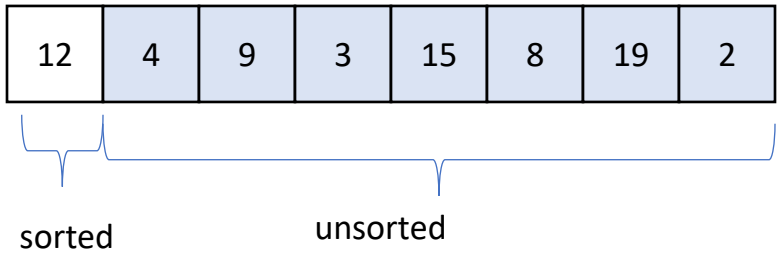


https://www.youtube.com/watch?v=ROalU379l3U

Insertion sort: Worst case analysis

1st iter 2nd iter 3rd iter

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



Insertion sort: Best case analysis

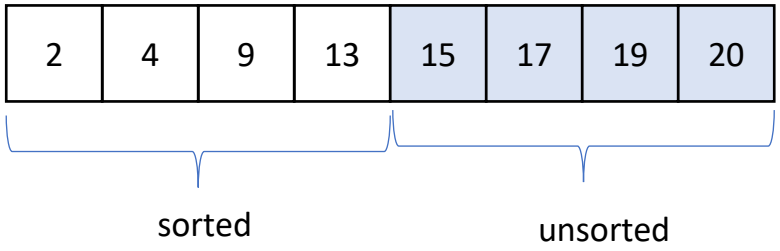
- Approximately how many steps does it take to insert the element into the sorted part each time through the loop in the BEST case?

A. 1

B. N

C. N^2

D. It depends on the length of the sorted part



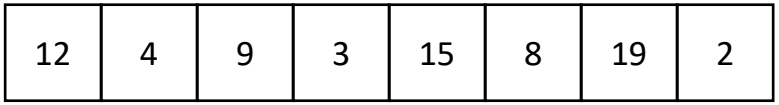
Array Sorting Algorithms (wiki)

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))

MergeSort: The Magic of Recursion

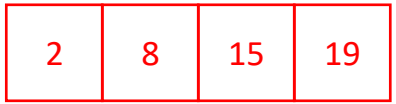
https://www.youtube.com/watch?v=XaqR3G_NVoo

- Consider this magical way of sorting lists:



Split the list in half:

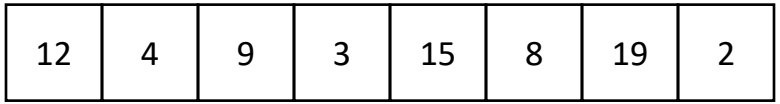
Magically sort each list



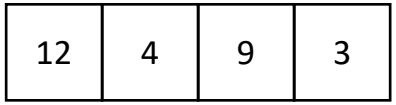
Merge the two lists back together

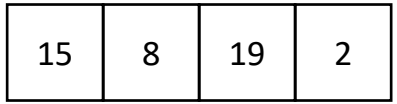
MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:

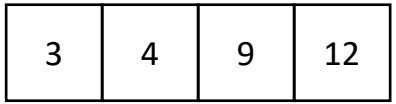


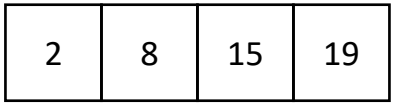
Split the list in half:





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

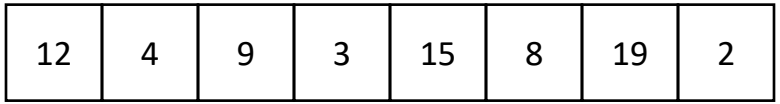




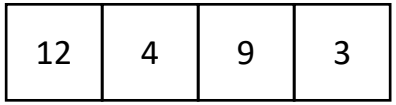
Merge the two lists back together

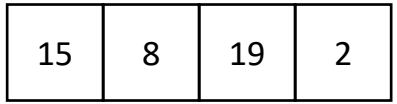
MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:

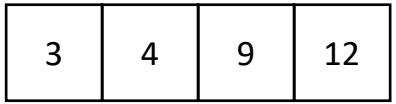


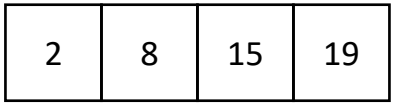
Split the list in half:





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 [a, b)

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

Does this mergeSort method work?

A. Yes
B. No

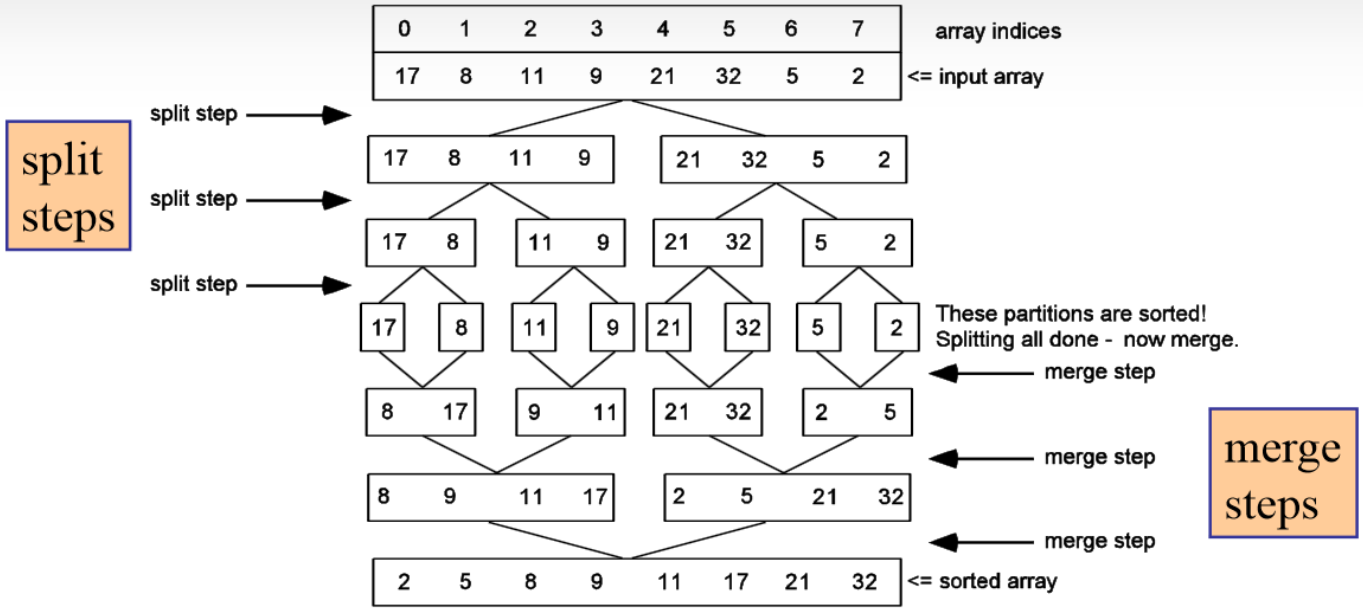
```
public void mergeSort( int[] toSort )
{
  if (toSort.length > 1) {
    int mid = toSort.length / 2;
    int[] firstHalf = makeList(toSort, 0, mid);
    int[] secondHalf = makeList( toSort, mid, toSort.length );
    mergeSort( firstHalf );
    mergeSort( secondHalf );
    merge( firstHalf, secondHalf,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

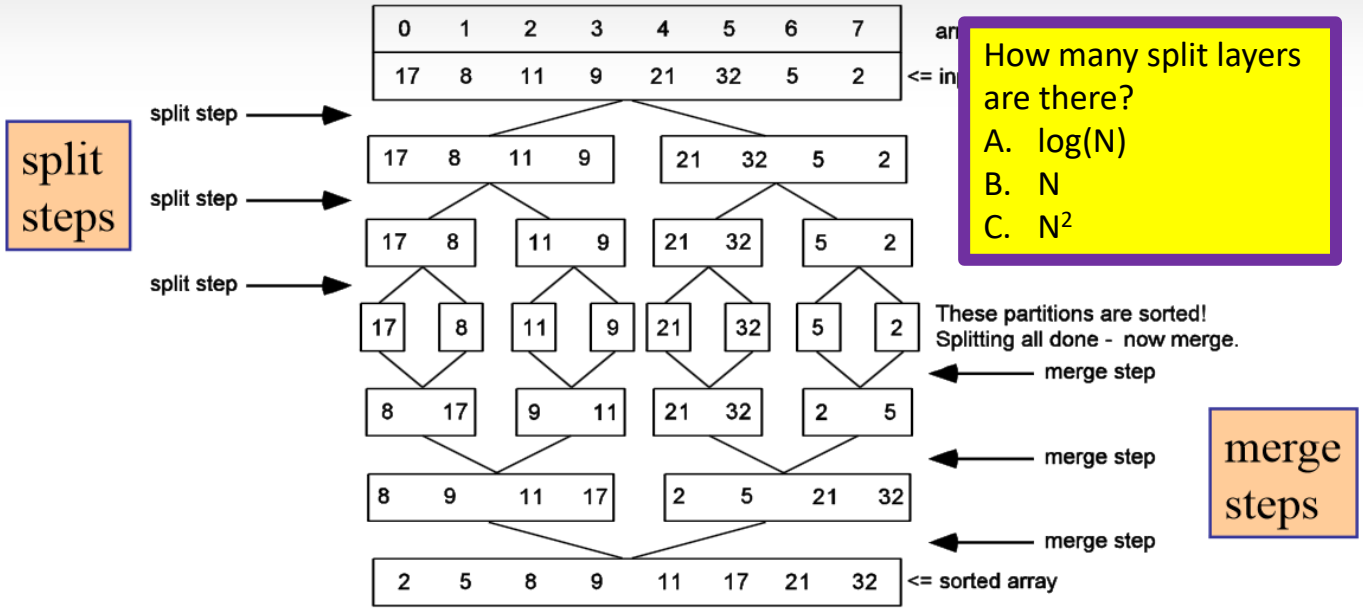
This mergeSort works!!

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

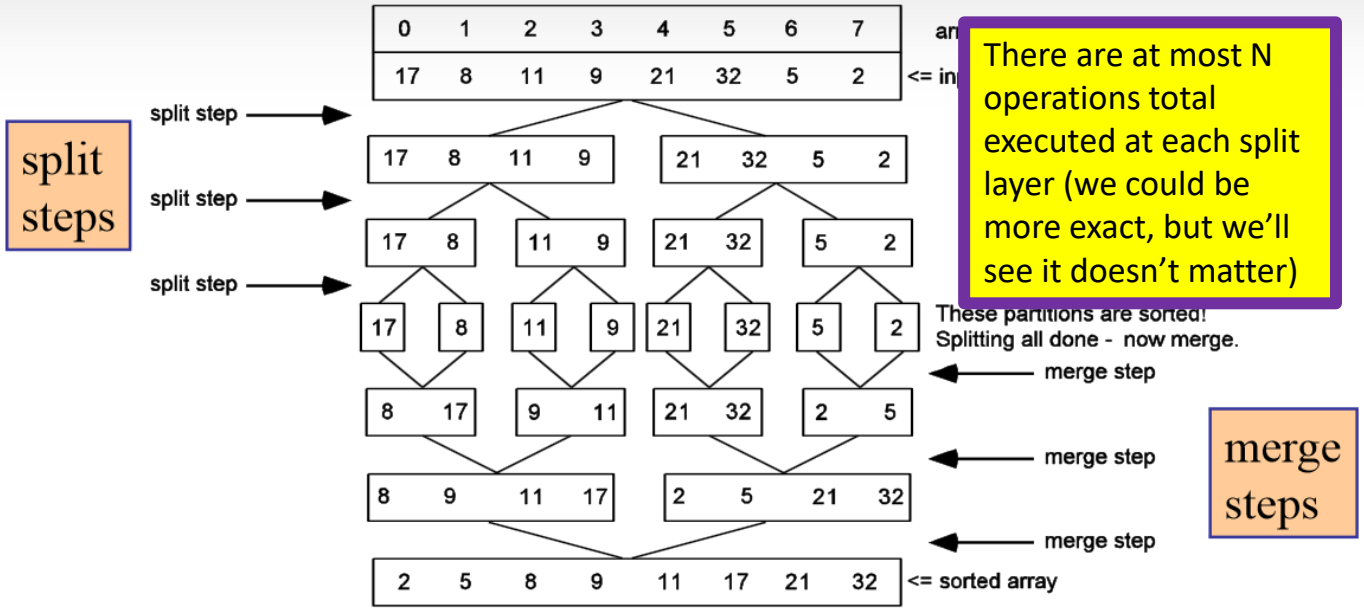


How many split layers are there?

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

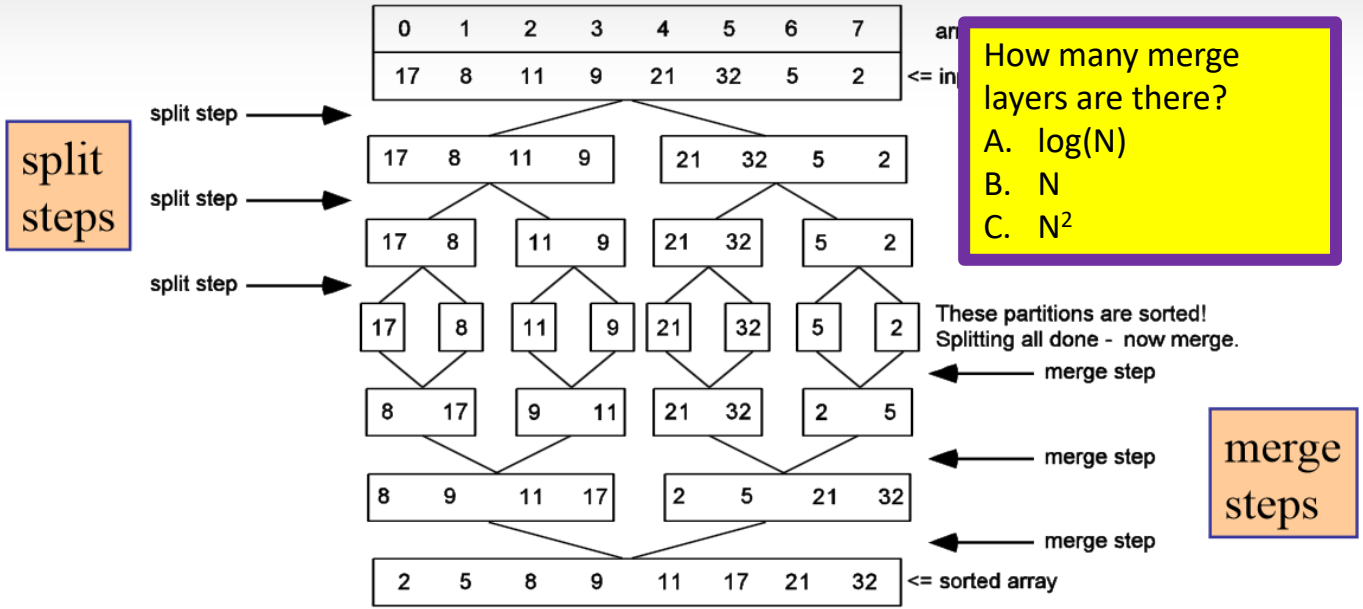
Merge Sort: An Example



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

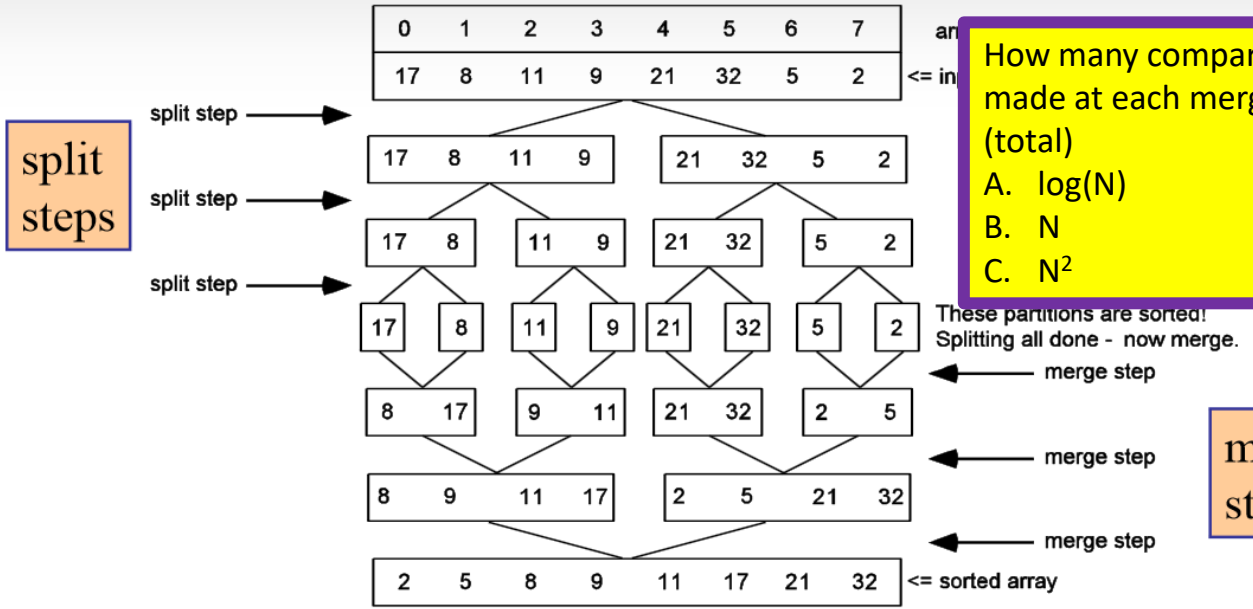


How many merge layers are there?

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

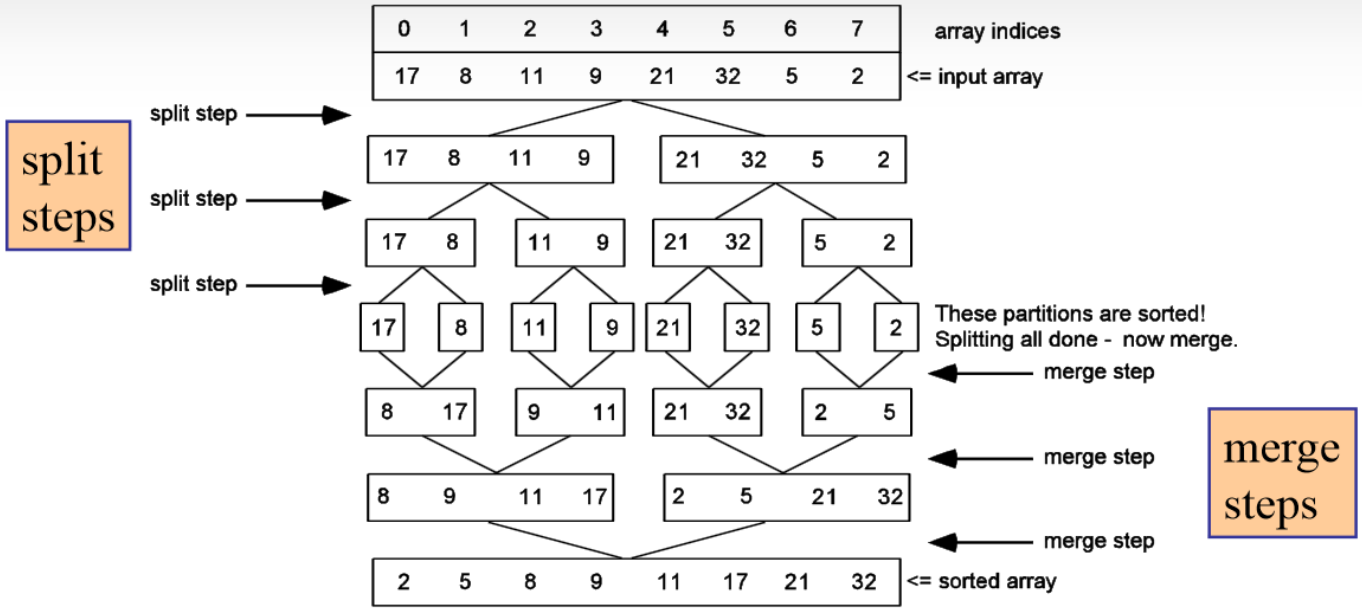


How many comparisons are made at each merge layer? (total)

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))

```
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));}
```