# ECS32B

## Introduction to Data Structures

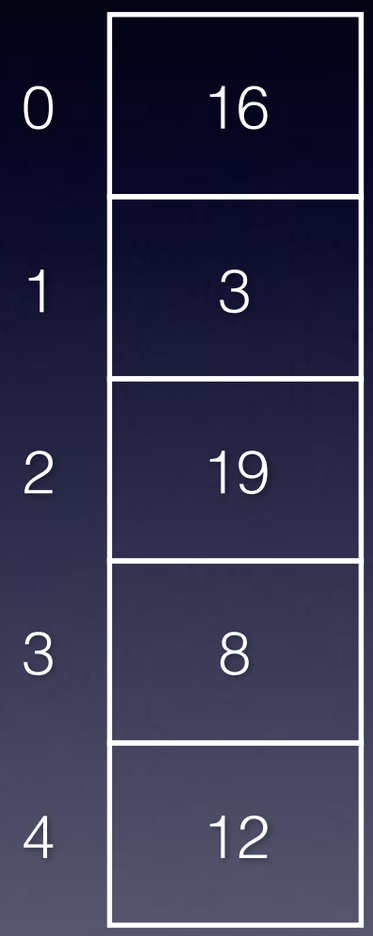
## Sorting

## Lecture 20

# Announcements

- Homework 4 due tonight at 11:59pm
- Zheng will be in Hutch 78B from 4:30-6:30
- Homework 5 will be posted tonight.

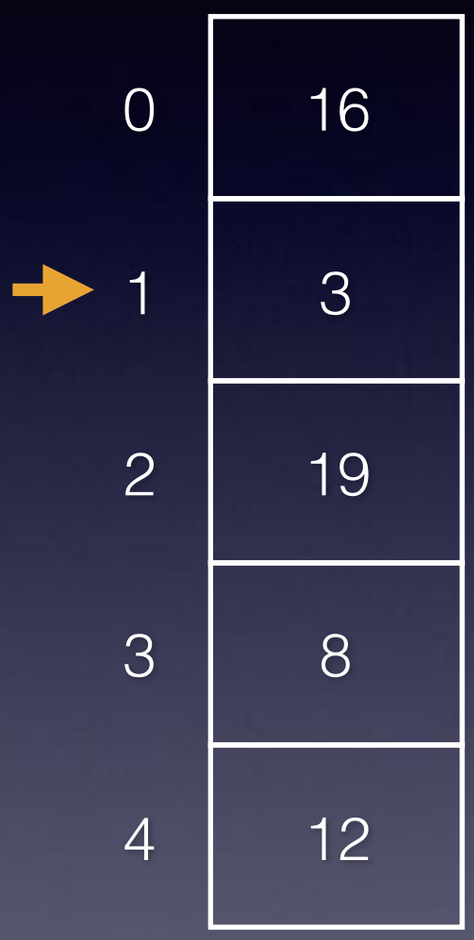
# Insertion sort




Insertion sort takes a slightly different approach compared to selection sort.

Let's assume that when we begin to sort the elements of a list, the very first element represents a sorted list.  Everything after that remains to be sorted.

# Insertion sort



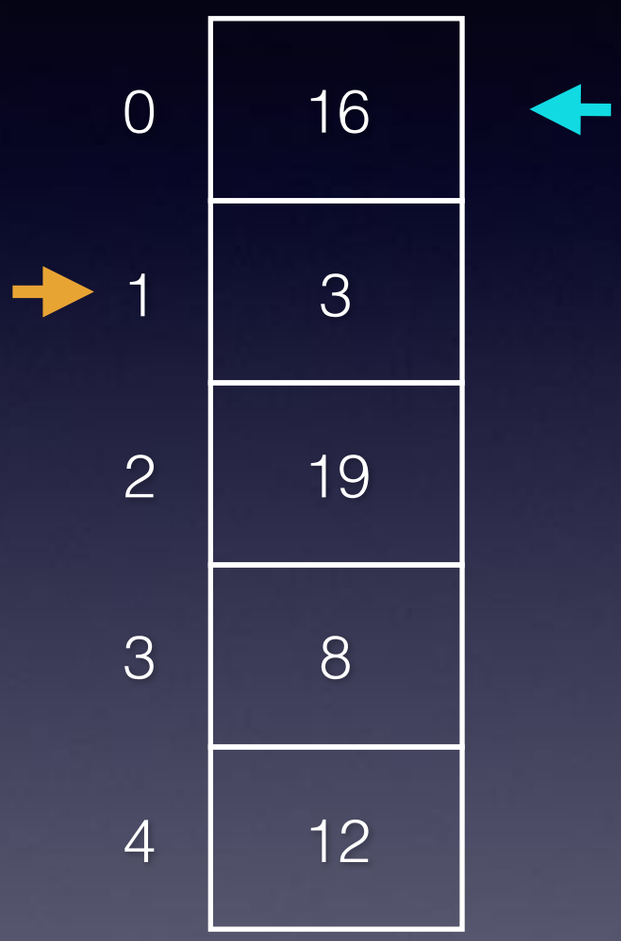

The value to be inserted is 3.

In this example our sorted portion of the list is the value 16.

The next value to be inserted in the sorted portion is 3, it is the first value in the unsorted portion (indicated by the **left** arrow)

We **store** the value indicated by the **left** arrow as the item to be inserted into the sorted portion.

# Insertion sort

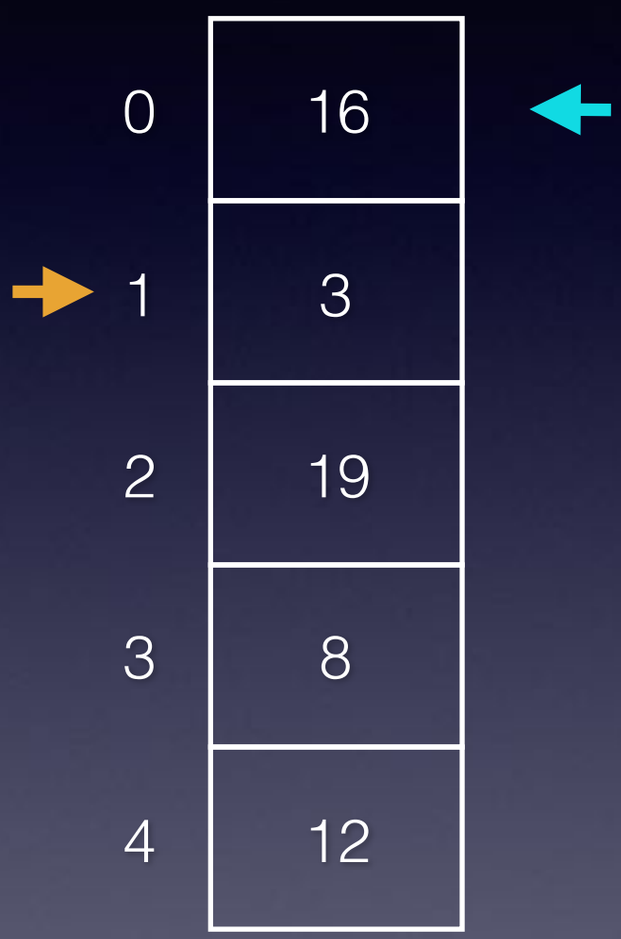



The value to be inserted is 3.

Now we start comparing the value to be inserted to the sorted items above it, from bottom to top, to determine where the new item belongs among the already sorted items.

We'll indicate the bottom of the sorted portion with the **right** arrow.

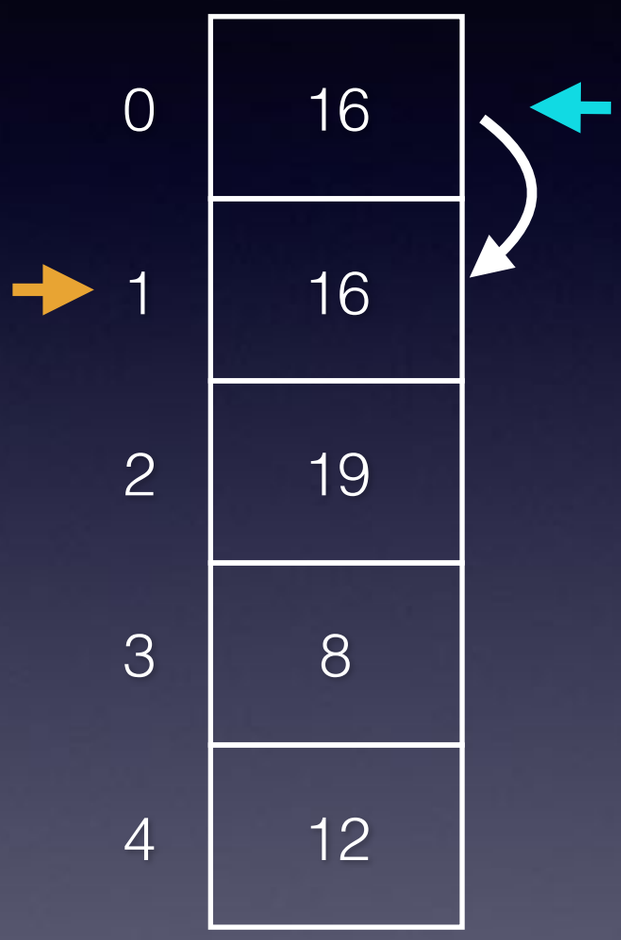
# Insertion sort




We ask “Is 3 less than 16, the value at the bottom of the sorted portion?”

The value to be inserted is 3.

# Insertion sort

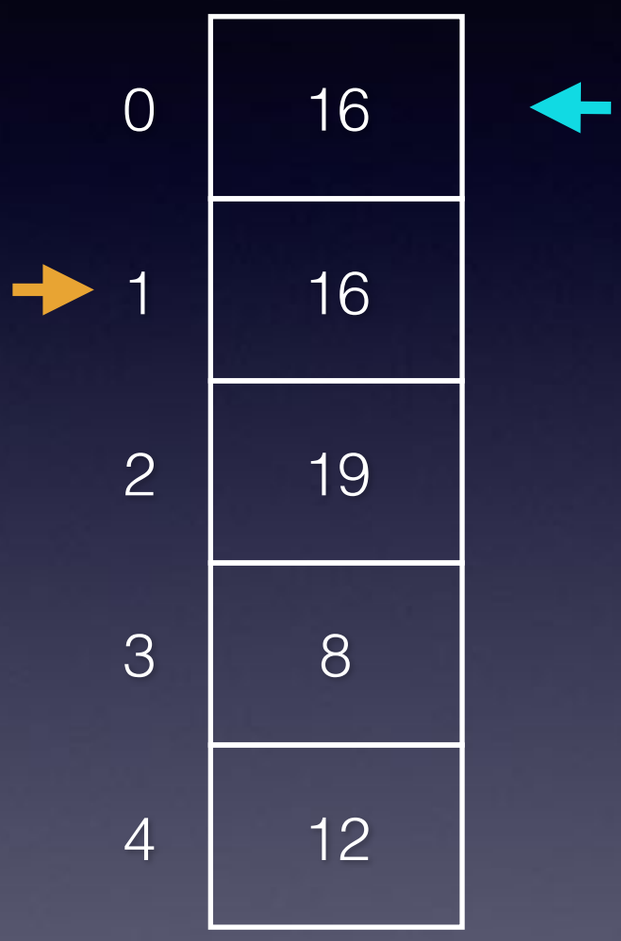



The value to be inserted is 3.

We ask “Is 3 less than 16, the value at the bottom of the sorted portion?”

If so, 3 needs to be above 16 in the sorted portion.  We have to make space for 3, so we move 16 down one space.

# Insertion sort



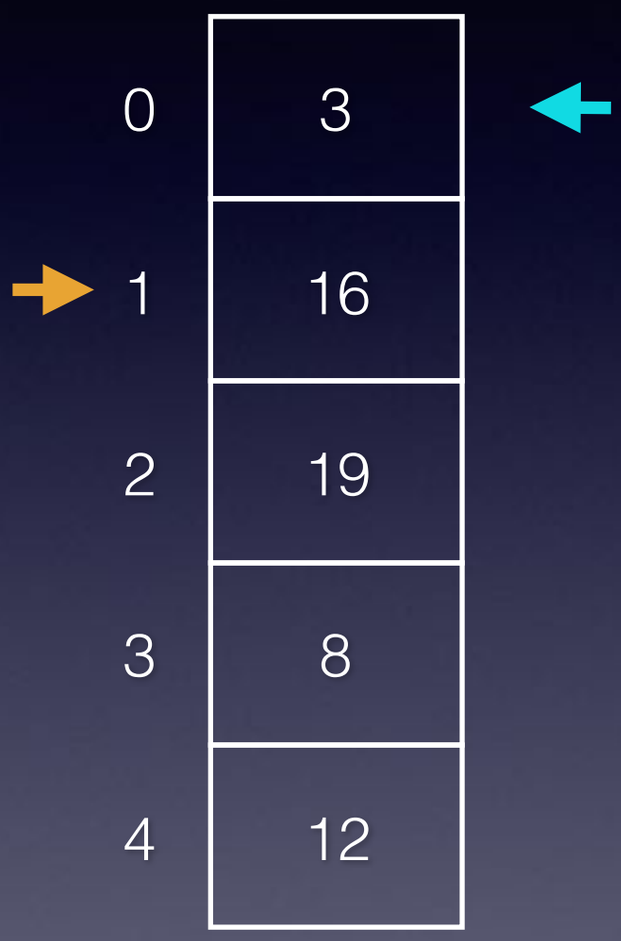

The value to be inserted is 3.

We ask “Is 3 less than 16, the value at the bottom of the sorted portion?”

If so, 3 needs to be above 16 in the sorted portion.  We have to make space for 3, so we move 16 down one space.

We want to look at the next to last item in the sorted portion, but there is nothing to look at. We’ve run out of list, so we copy the item to be inserted into the top or first place in the list.

# Insertion sort



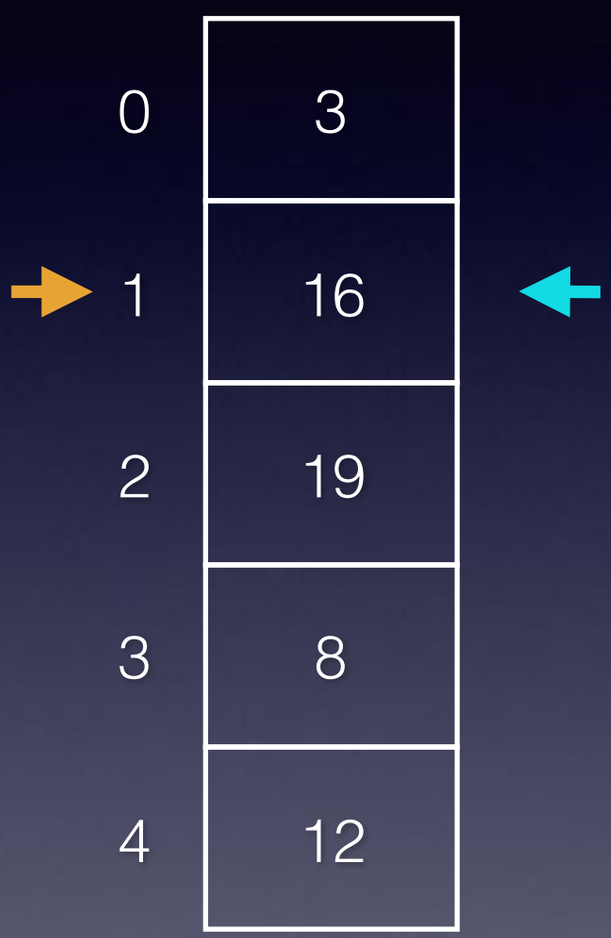

The value to be inserted is 3.

We ask “Is 3 less than 16, the value at the bottom of the sorted portion?”

If so, 3 needs to be above 16 in the sorted portion.  We have to make space for 3, so we move 16 down one space.

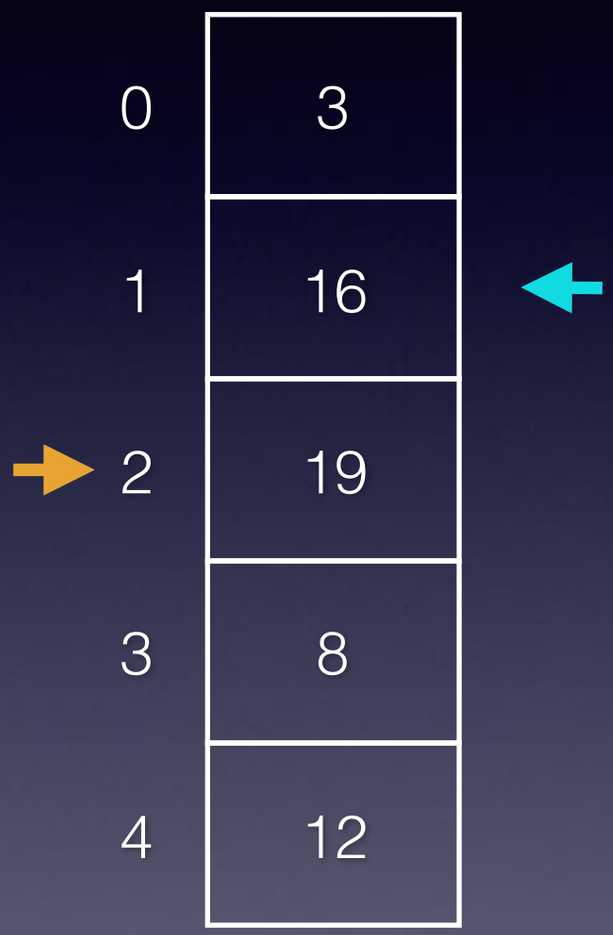
We want to look at the next to last item in the sorted portion, but there is nothing to look at. We’ve run out of list, so we copy the item to be inserted into the top or first place in the list.

# Insertion sort




We reset the **right** pointer to the bottom of the sorted portion.

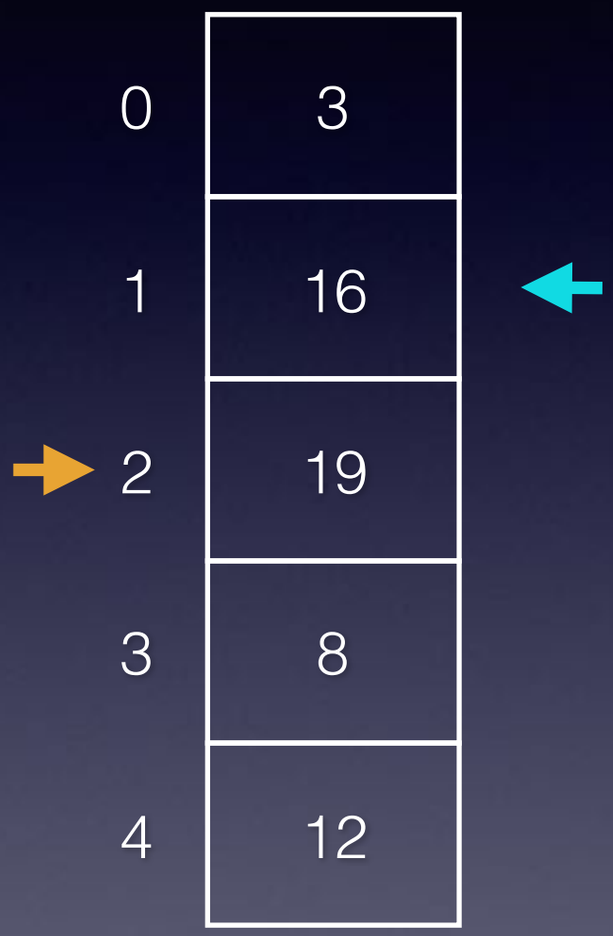
# Insertion sort




We reset the **right** pointer to the bottom of the sorted portion.

We move the **left** arrow to the top of the the unsorted portion.

# Insertion sort



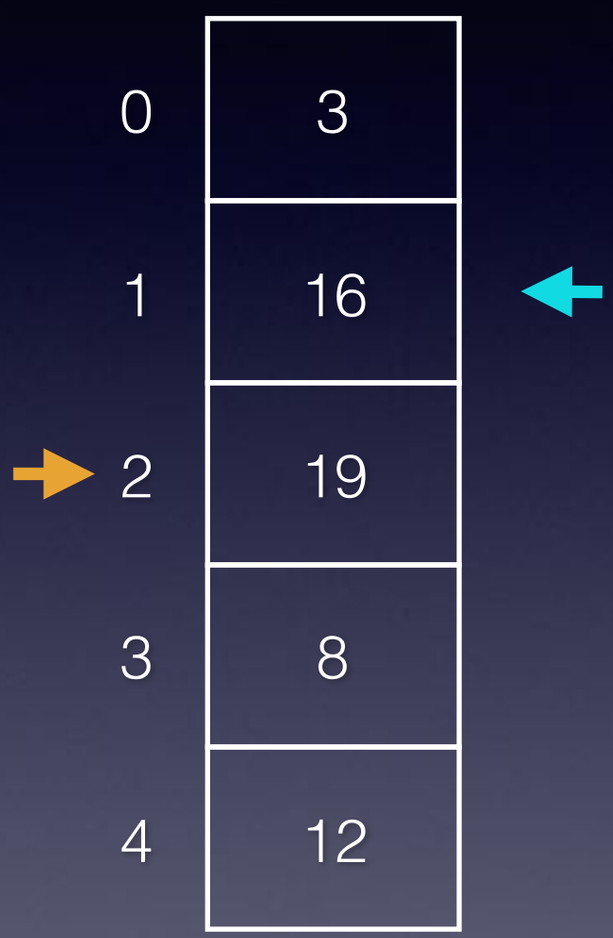

The value to be inserted is 19.

We reset the **right** pointer to the bottom of the sorted portion.

We move the **left** arrow to the top of the the unsorted portion.

We store the value indicated by the **left** arrow as the item to be inserted into the sorted portion.

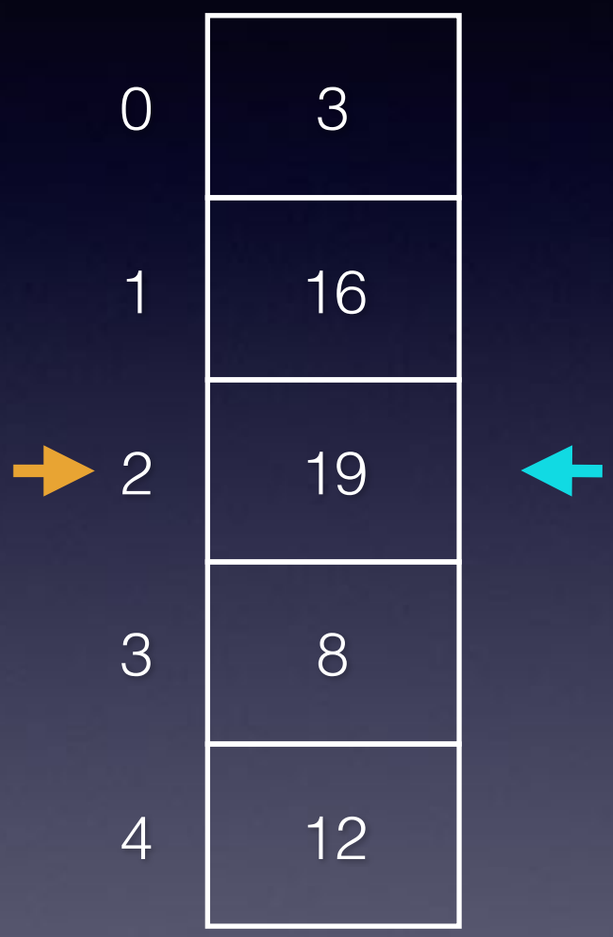
# Insertion sort




Is 19 less than 16? No. So we don't need to look any further; 19 is where it should be.

The value to be inserted is 19.

# Insertion sort

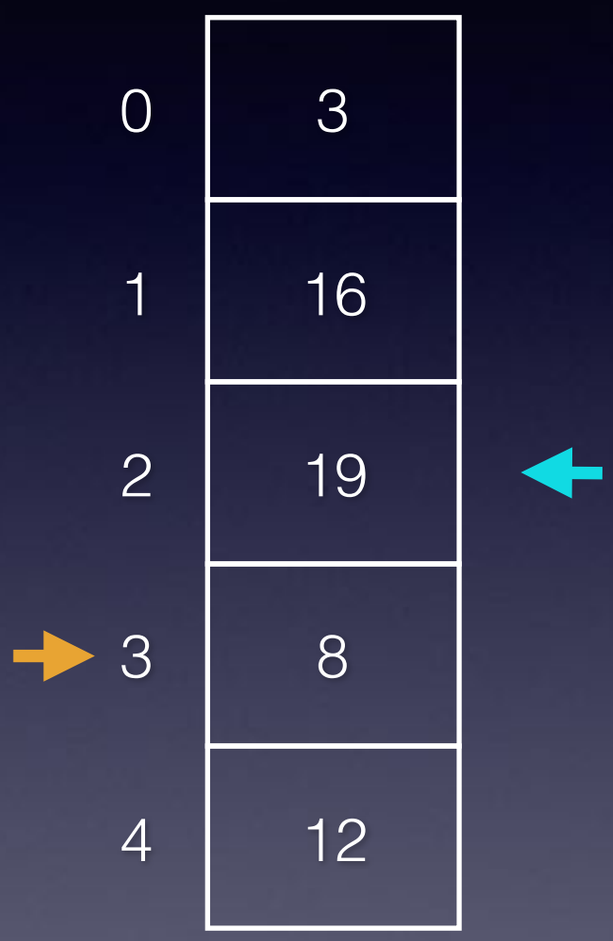



The value to be inserted is 19.

Is 19 less than 16? No. So we don't need to look any further; 19 is where it should be.

Now the top three elements are the sorted portion,

# Insertion sort



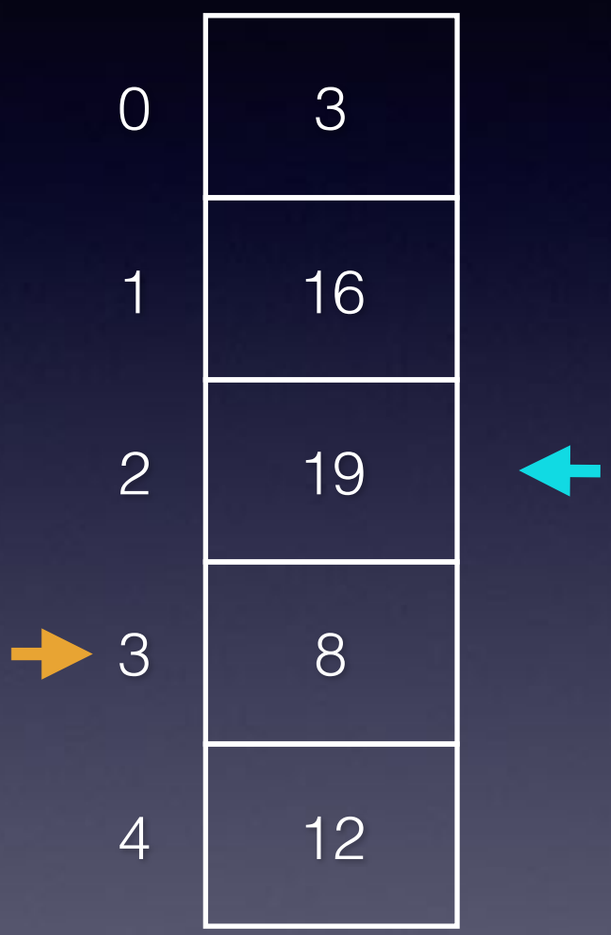

The value to be inserted is 8.

Is 19 less than 16? No. So we don't need to look any further; 19 is where it should be.

Now the top three elements are the sorted portion, and we store the next element to be inserted into the sorted portion.

# Insertion sort

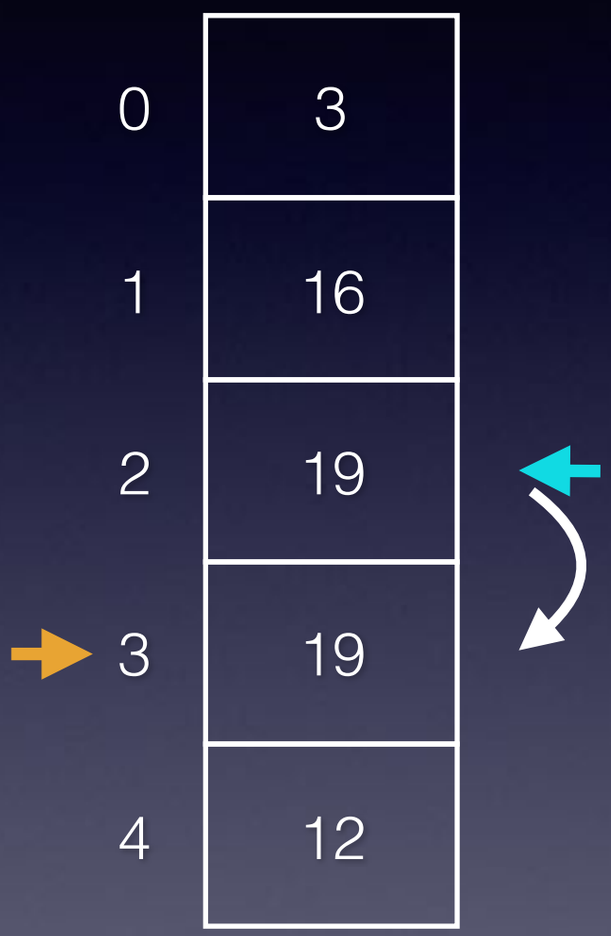
Is 8 less than 19?




The value to be inserted is 8.

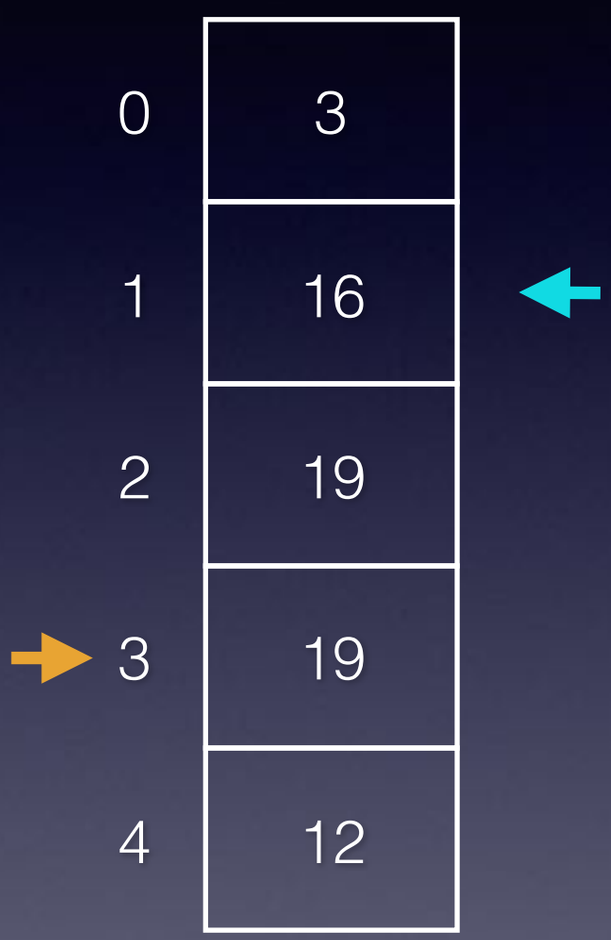
# Insertion sort

Is 8 less than 19?  Yes, so 19 moves down.




The value to be inserted is 8.

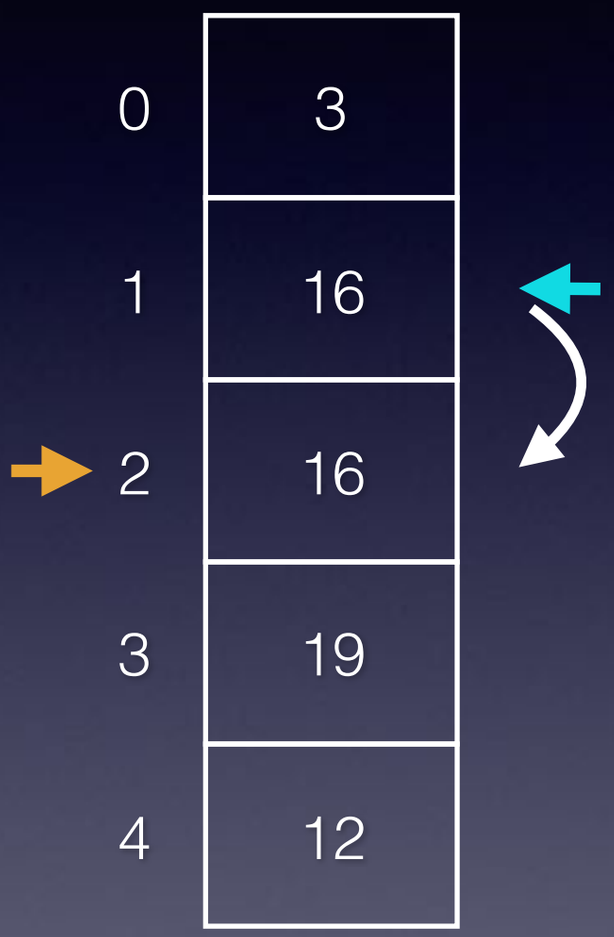
# Insertion sort




Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16?

The value to be inserted is 8.

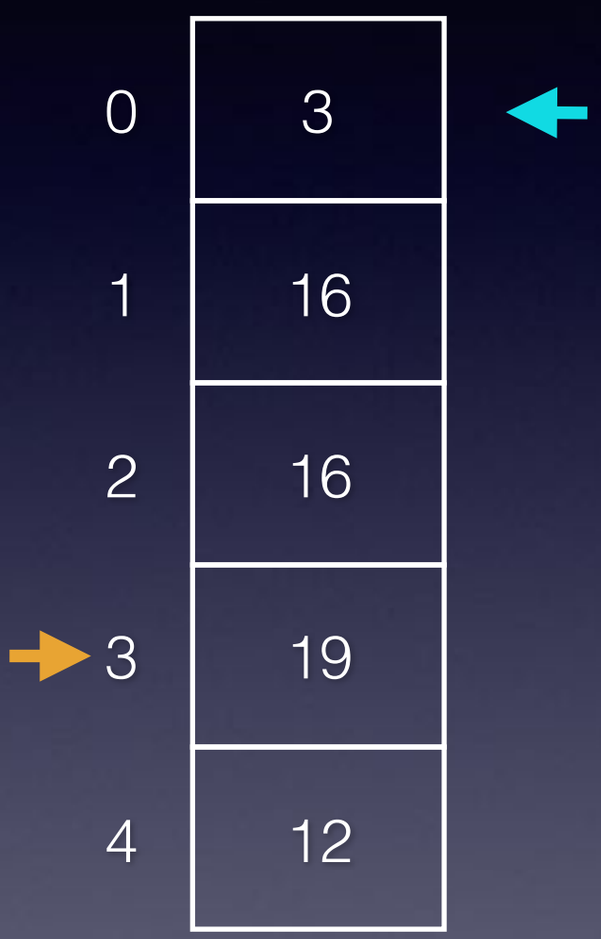
# Insertion sort




Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.

The value to be inserted is 8.

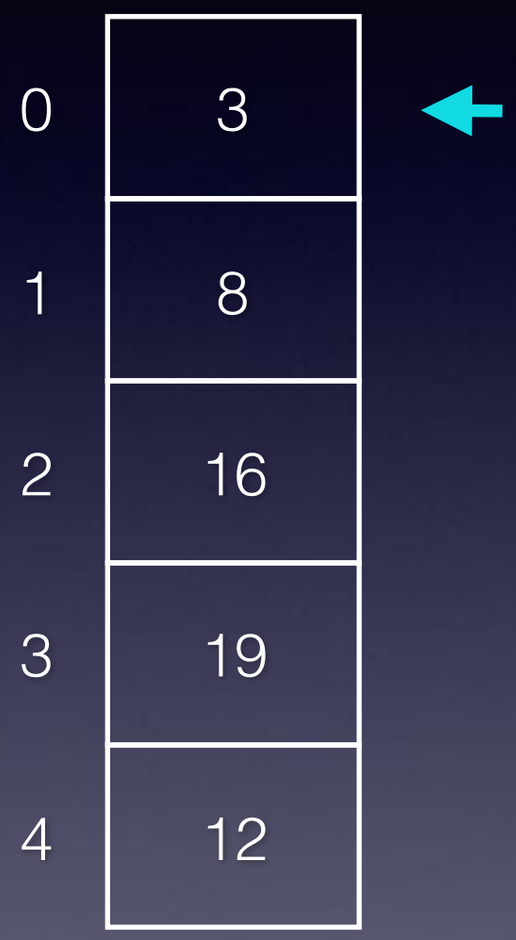
# Insertion sort




Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.
Is 8 less than 3?

The value to be inserted is 8.

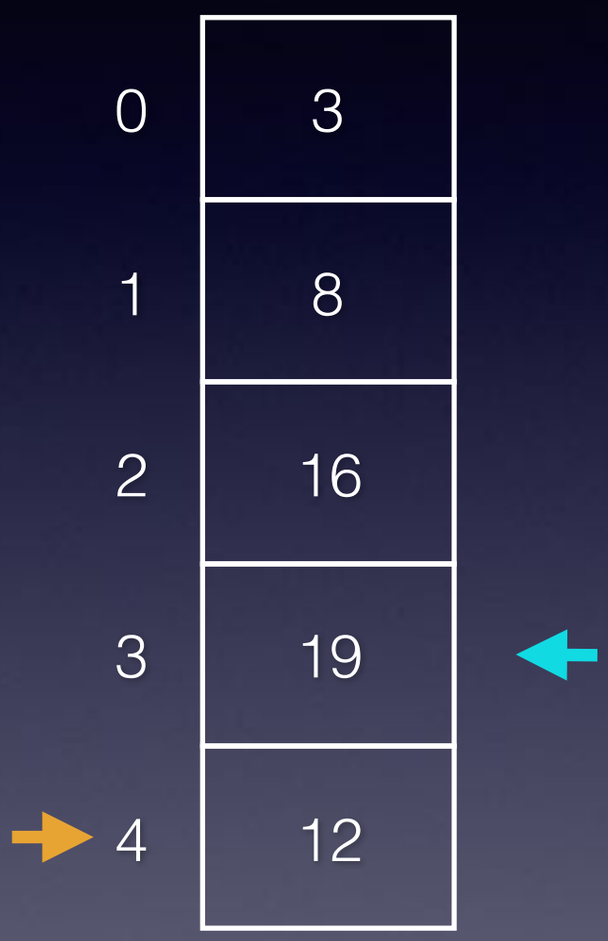
# Insertion sort




The value to be inserted is 8.

Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.
Is 8 less than 3? No, so we copy 8 into the space that was last vacated (i.e., the empty space that's waiting to hold the value to be inserted).

# Insertion sort

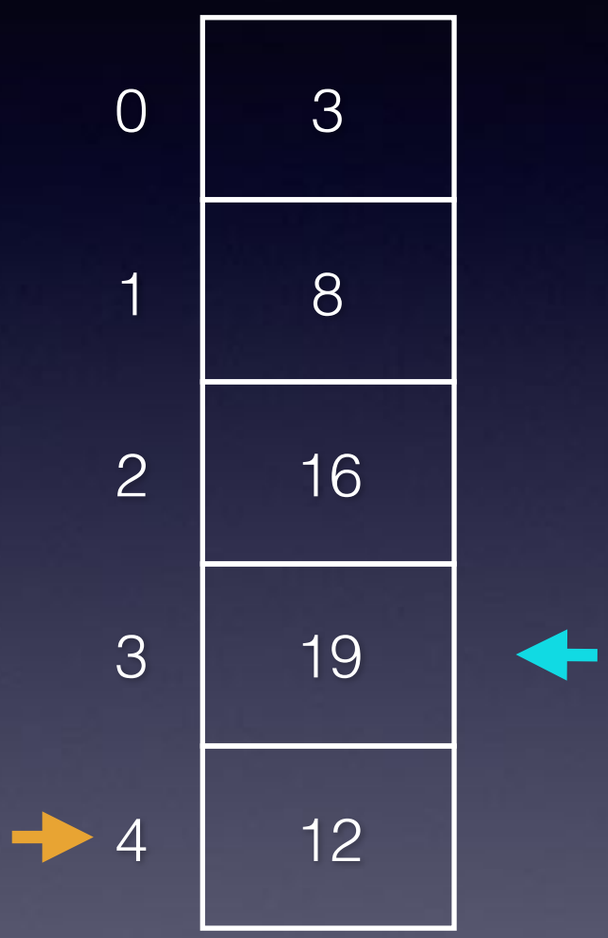



The value to be inserted is 12.

Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.
Is 8 less than 3? No, so we copy 8 into the space that was last vacated (i.e., the empty space that's waiting to hold the value to be inserted).

Reset the bottom of the sorted portion and the next value to be inserted.

# Insertion sort

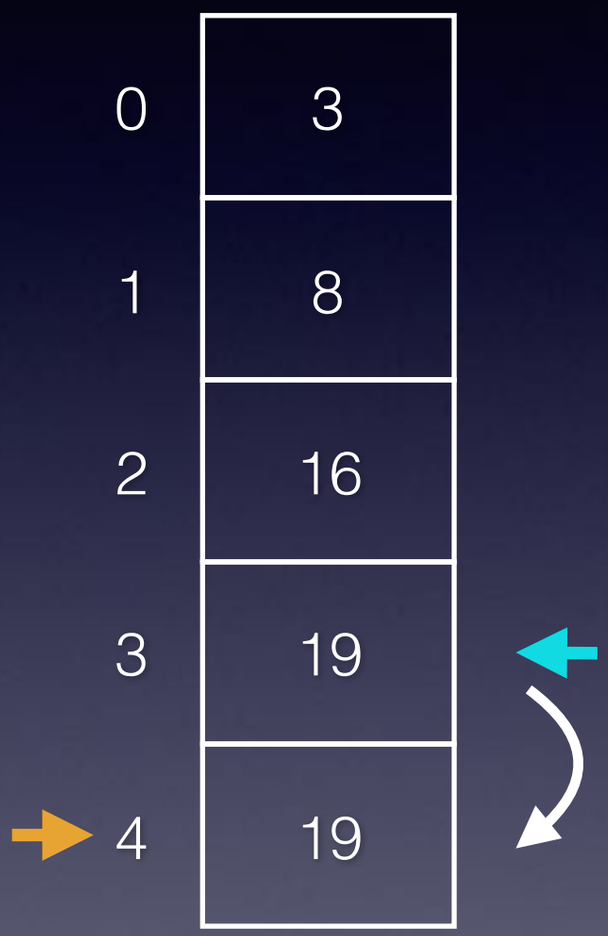



Is 12 less than 19?

The value to be inserted is 12.

# Insertion sort

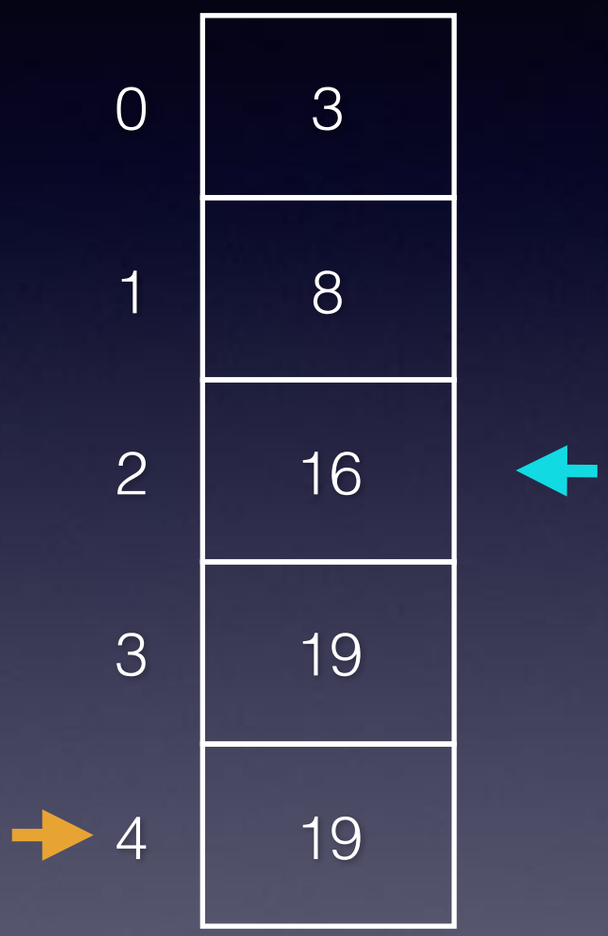
Is 12 less than 19? Yes, so move 19 down




The value to be inserted is 12.

# Insertion sort

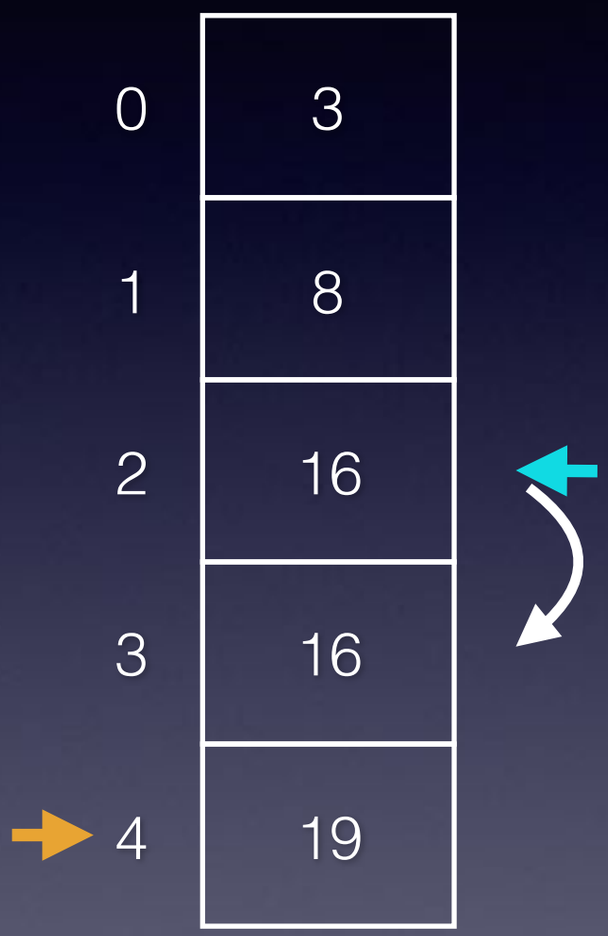
Is 12 less than 16?




The value to be inserted is 12.

# Insertion sort

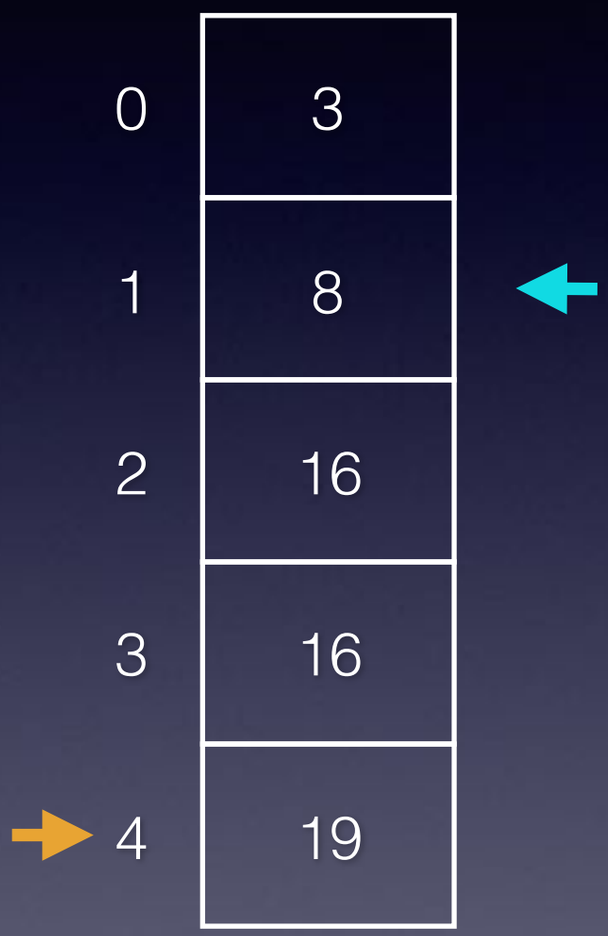
Is 12 less than 16? Yes, so 16 moves down.




The value to be inserted is 12.

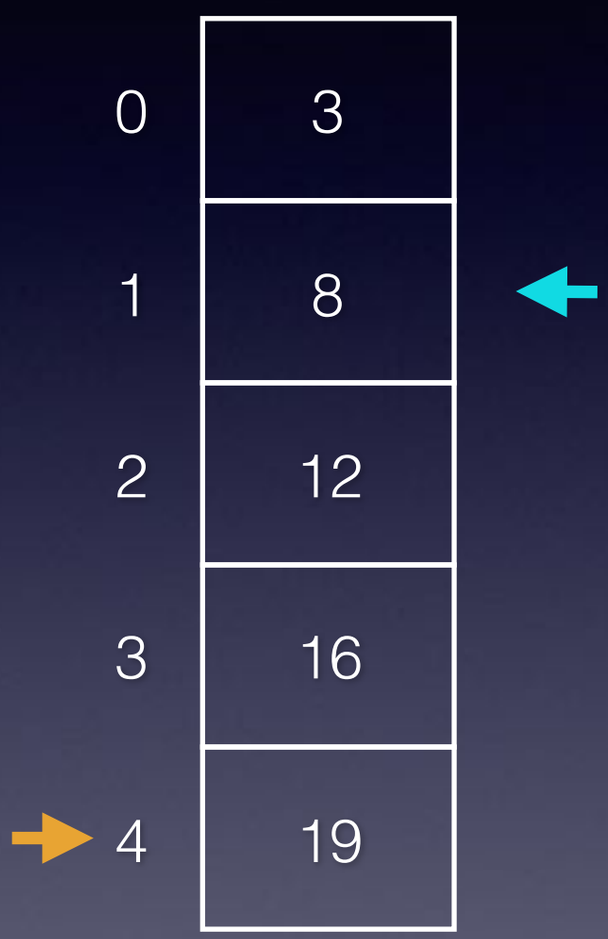
# Insertion sort

Is 12 less than 8?




The value to be inserted is 12.

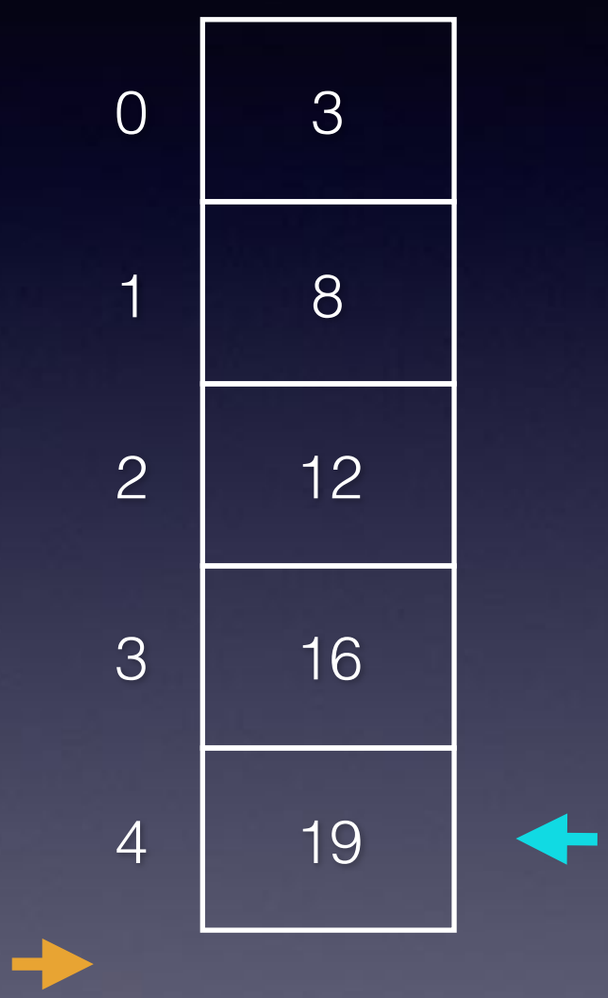
# Insertion sort




Is 12 less than 8? No, so we copy 12 into the last vacated space.

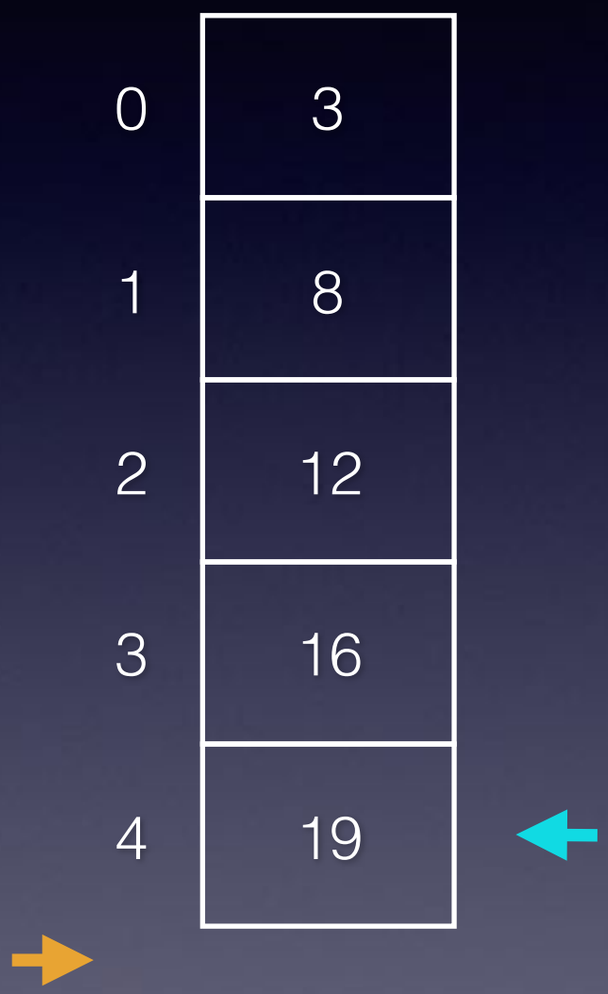
The value to be inserted is 12.

# Insertion sort




When we try to reset the pointers now, we can see that we're done.  The entire list is sorted.

# Insertion sort



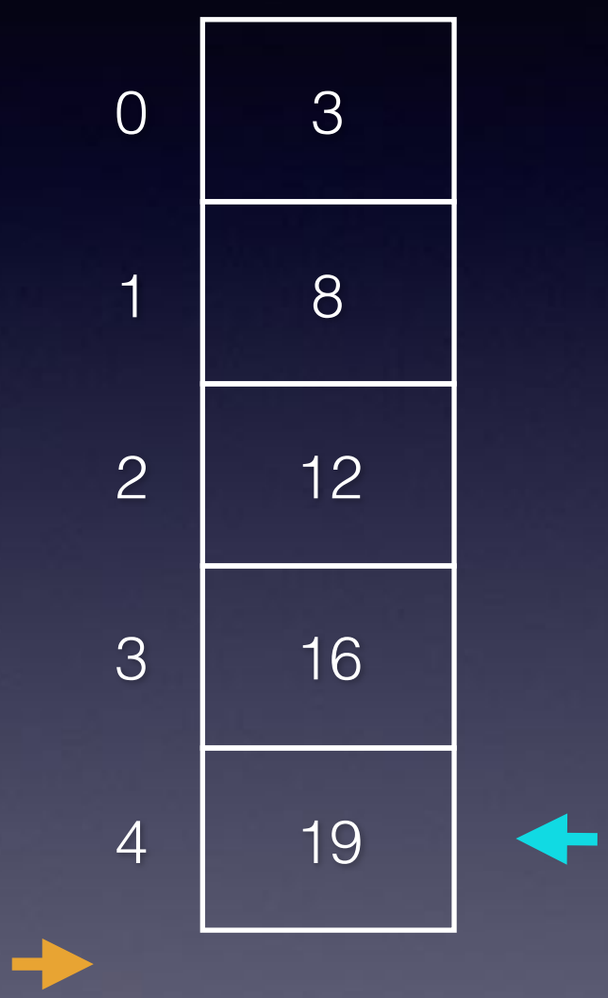

What input would let insertion sort show off its best possible performance?

A list that's already sorted.

How many comparisons would be made?

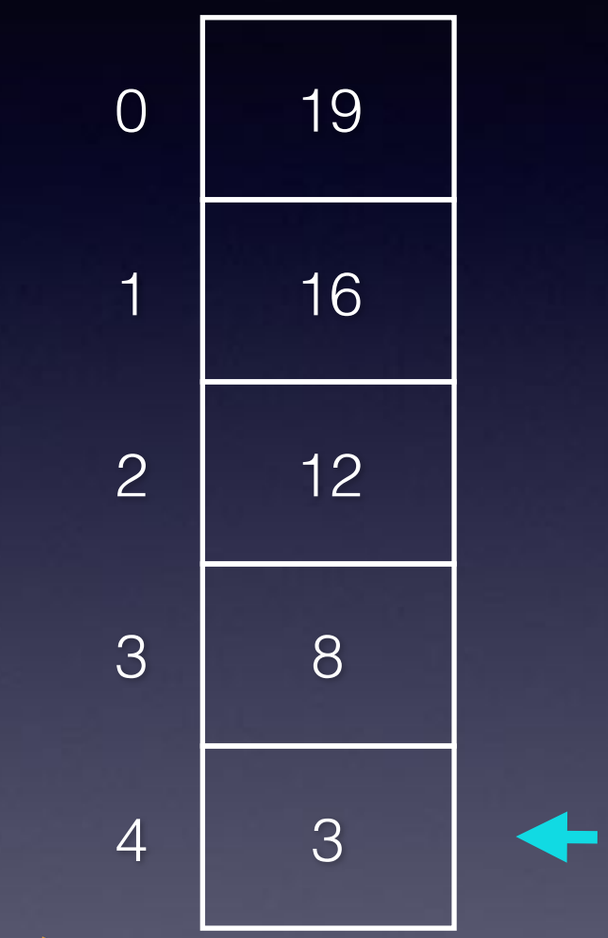
How many elements would be moved?

# Insertion sort




What input would be the worst possible case for insertion sort?

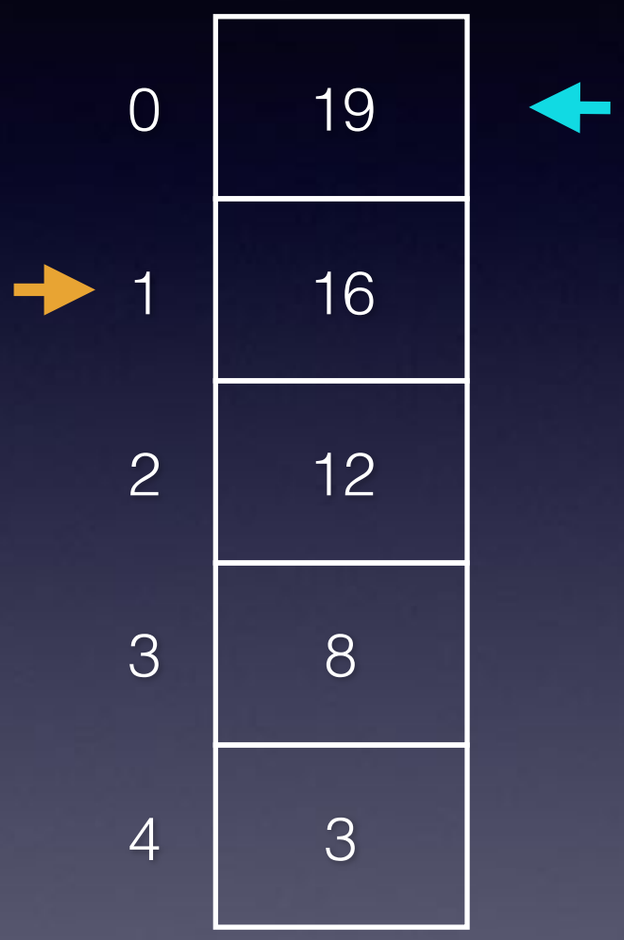
# Insertion sort




What input would be the worst possible case for insertion sort?

Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

# Insertion sort

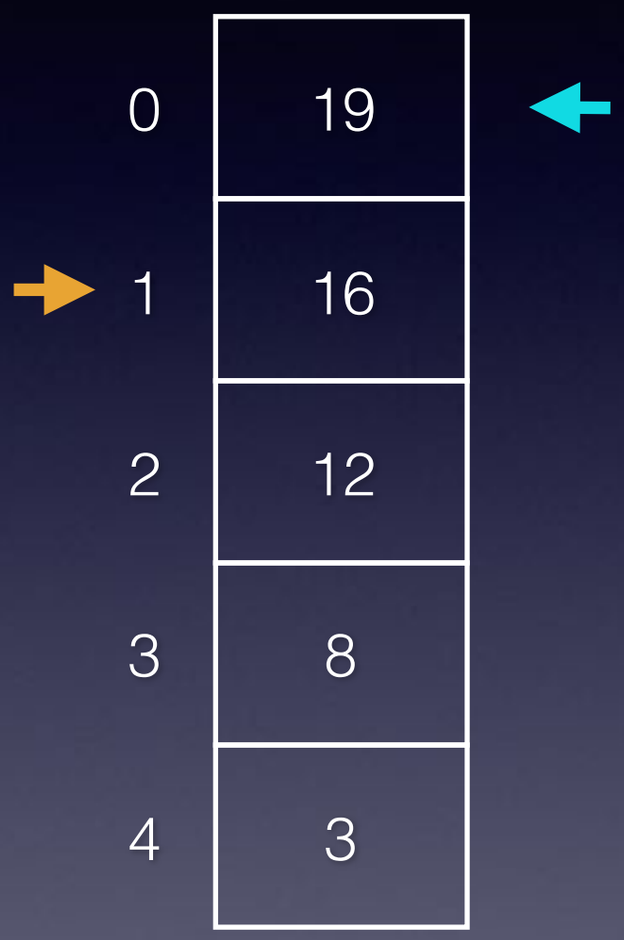



What input would be the worst possible case for insertion sort?

Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

So what's the worst case Big-O for insertion sort?

# Insertion sort

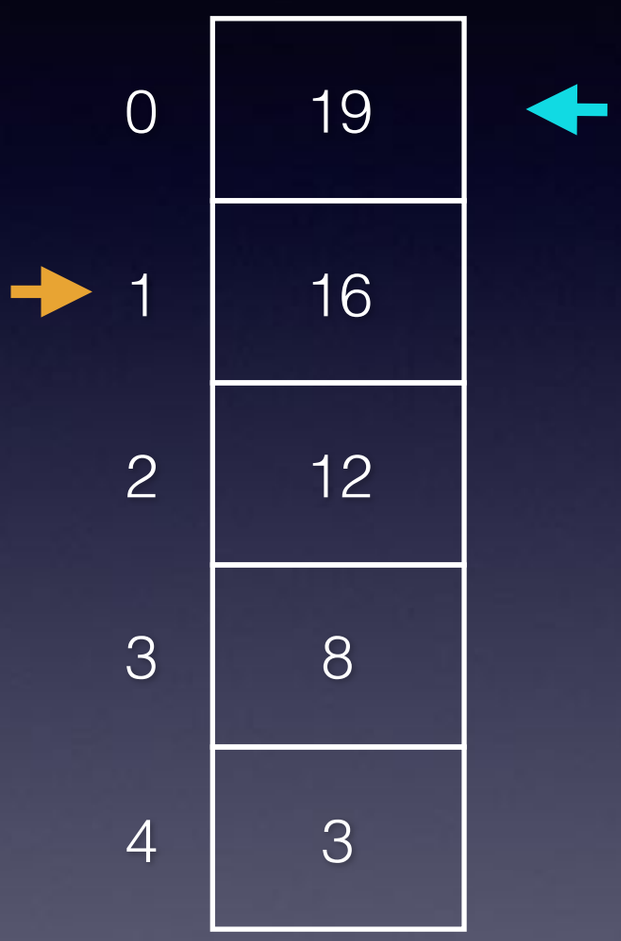



What input would be the worst possible case for insertion sort?

Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

So what's the worst case Big-O for insertion sort? $O(n^2)$, or pretty much the same as for selection sort.

# Insertion sort

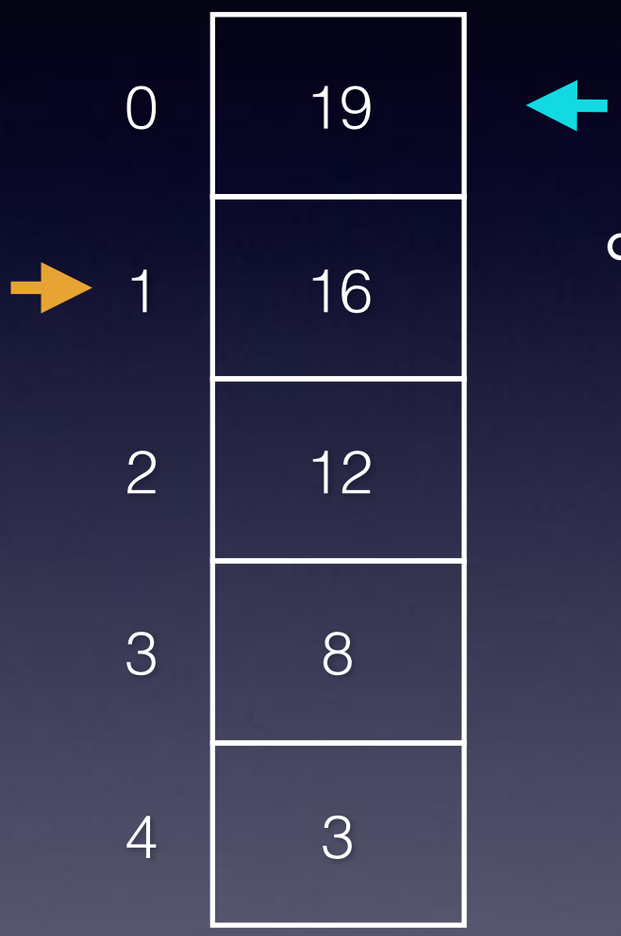



Pseudocode algorithm:

for each list element from the second
element to the last element
insert the selected element where it
belongs in the list by
shifting all values larger than the
selected element back by one location

Your textbook provides an insertionSort function that behaves like what we just discussed. It's on the next slide...

# Insertion sort




```
def insertionSort(alist):
    for index in range(1,len(alist)):
        currval = alist[index]
        pos = index
        while pos > 0 and alist[pos - 1] > currval:
            alist[pos] = alist[pos - 1]
            pos = pos – 1
        alist[pos] = currval
```

# Can we get better than $O(n^2)$?

Selection sort and insertion sort both have worst case time complexity of $O(n^2)$.

For small n, that's perfectly acceptable.  When n is in the billions, $O(n^2)$ is unworkable.

How do we get to the promised O(nlogn) time complexity?

As we saw, that would be a huge advantage when sorting really large data sets.

# Mergesort

Mergesort takes a different approach to the problem.  It falls in the class of algorithms called “divide and conquer”.

In mergesort, the list is continually split in half by applying the algorithm recursively to each half, until the base case is reached.

The base case is the sorted list of one element.

The actual sorting happens during the merge phase.

Merging two sorted lists together can be done in O(n) time.

# Mergesort

A simple algorithm for mergesort is:

**mergesort**(unsorted_list)

> Divide the unsorted_list into **two** sublists of **half the size** of the unsorted_list call them **left** and **right**.
>
> Apply **mergesort** to the **left** and **right** sublists to sort them.
>
> Merge those two now-sorted sublists back into one sorted list.

# Mergesort

The Python functions below implement mergesort (this is not the textbook version, but may be easier to follow...).

```
def mergesort(mlist):
    if len(mlist) < 2:
        return mlist
    else:
        mid = len(mlist)//2
        return merge(mergesort(mlist[:mid]),mergesort(mlist[mid:]))

# merge two sorted lists
def merge(left, right):
    if left == []:
        return right
    elif right == []:
        return left
    elif left[0] < right[0]:
        return [left[0]] + merge(left[1:], right)
    else:
        return [right[0]] + merge(left, right[1:])
```

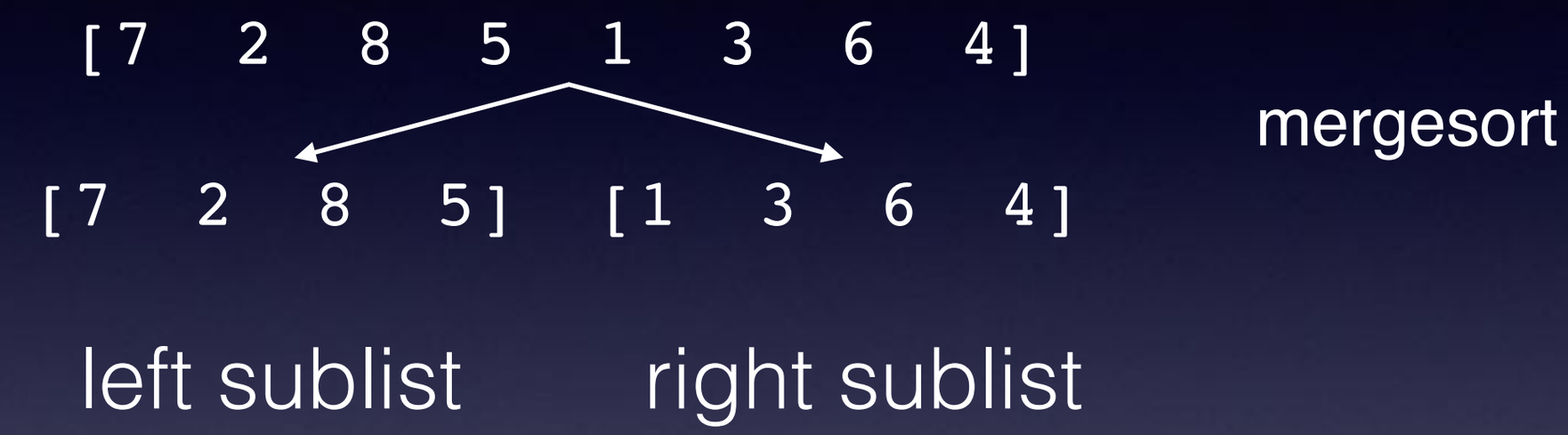
# Mergesort

Here's how these functions would sort a list of integers:

```
[7  2  8  5  1  3  6  4]
```

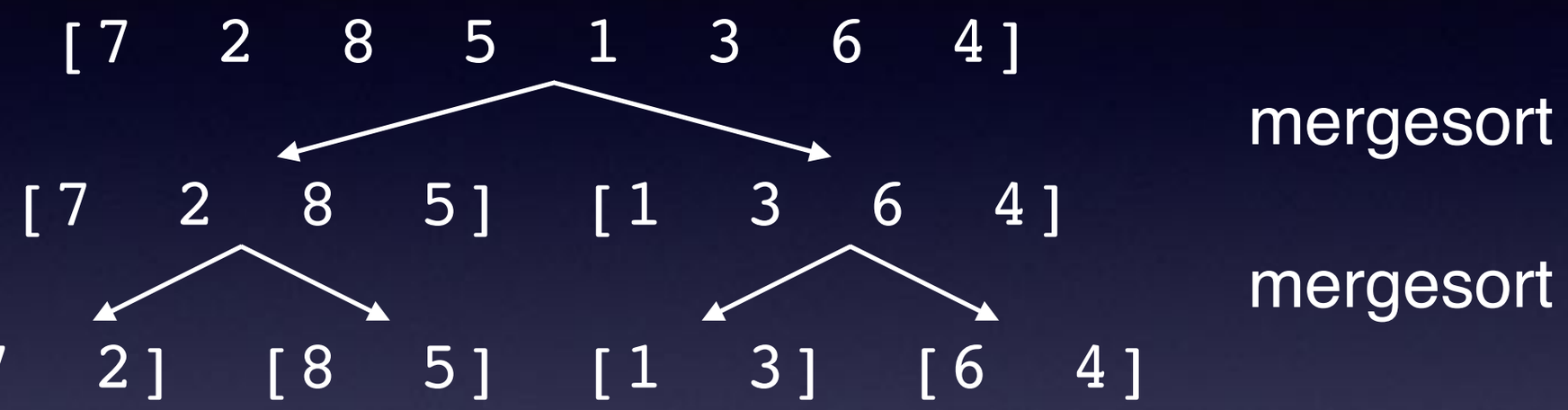
# Mergesort

Here's how these functions would sort a list of integers:



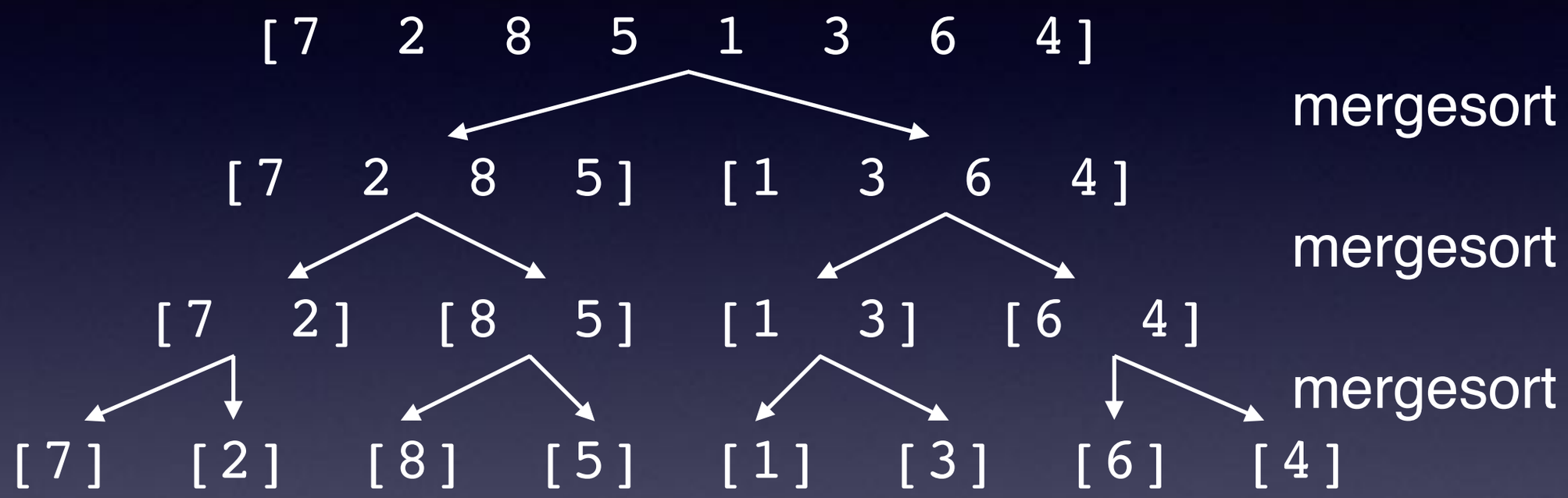
# Mergesort

Here's how these functions would sort a list of integers:



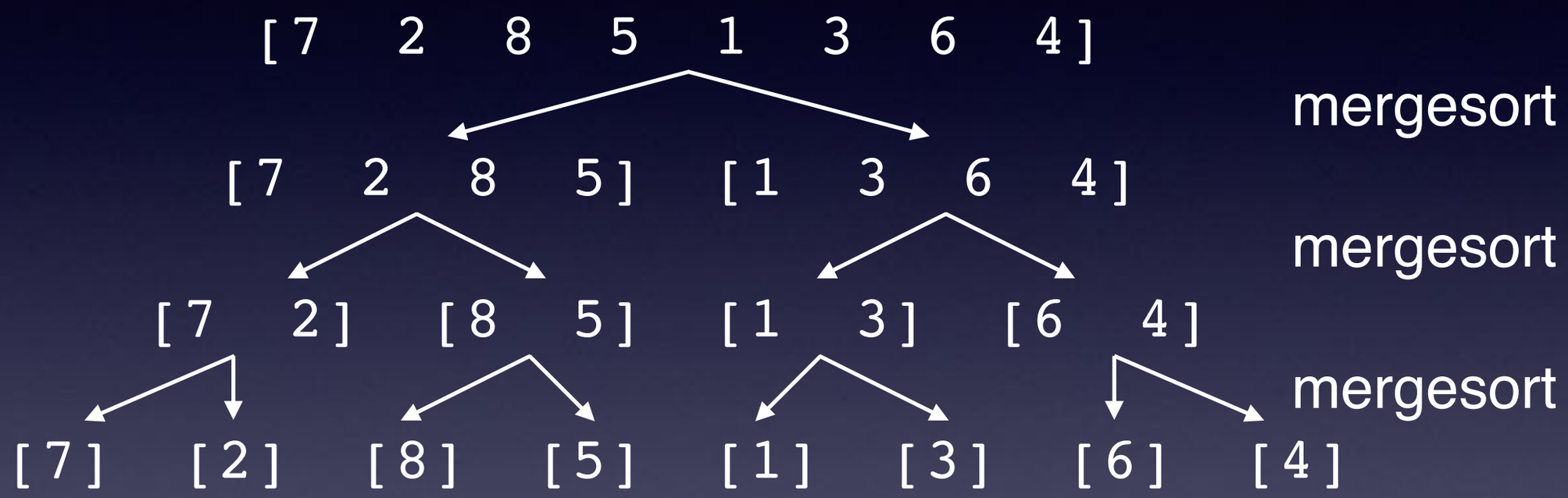
# Mergesort

Here's how these functions would sort a list of integers:



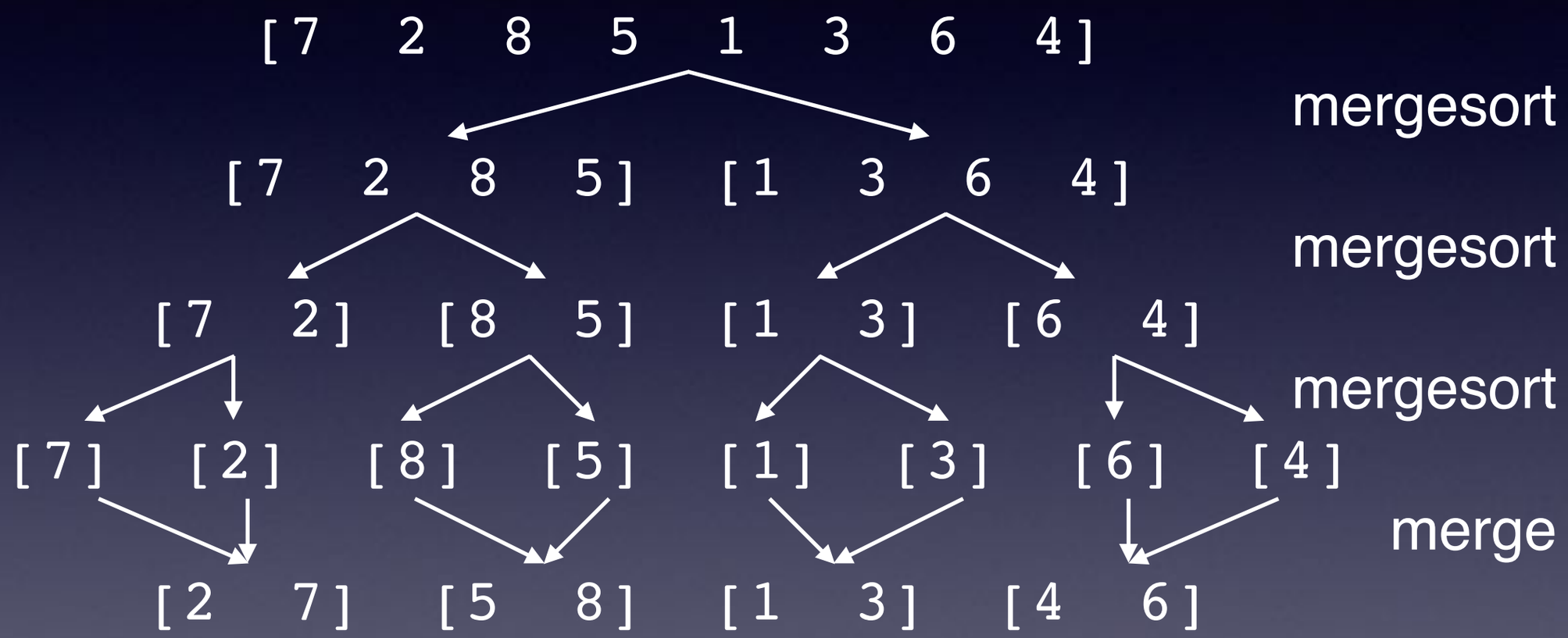
# Mergesort

Here's how these functions would sort a list of integers:



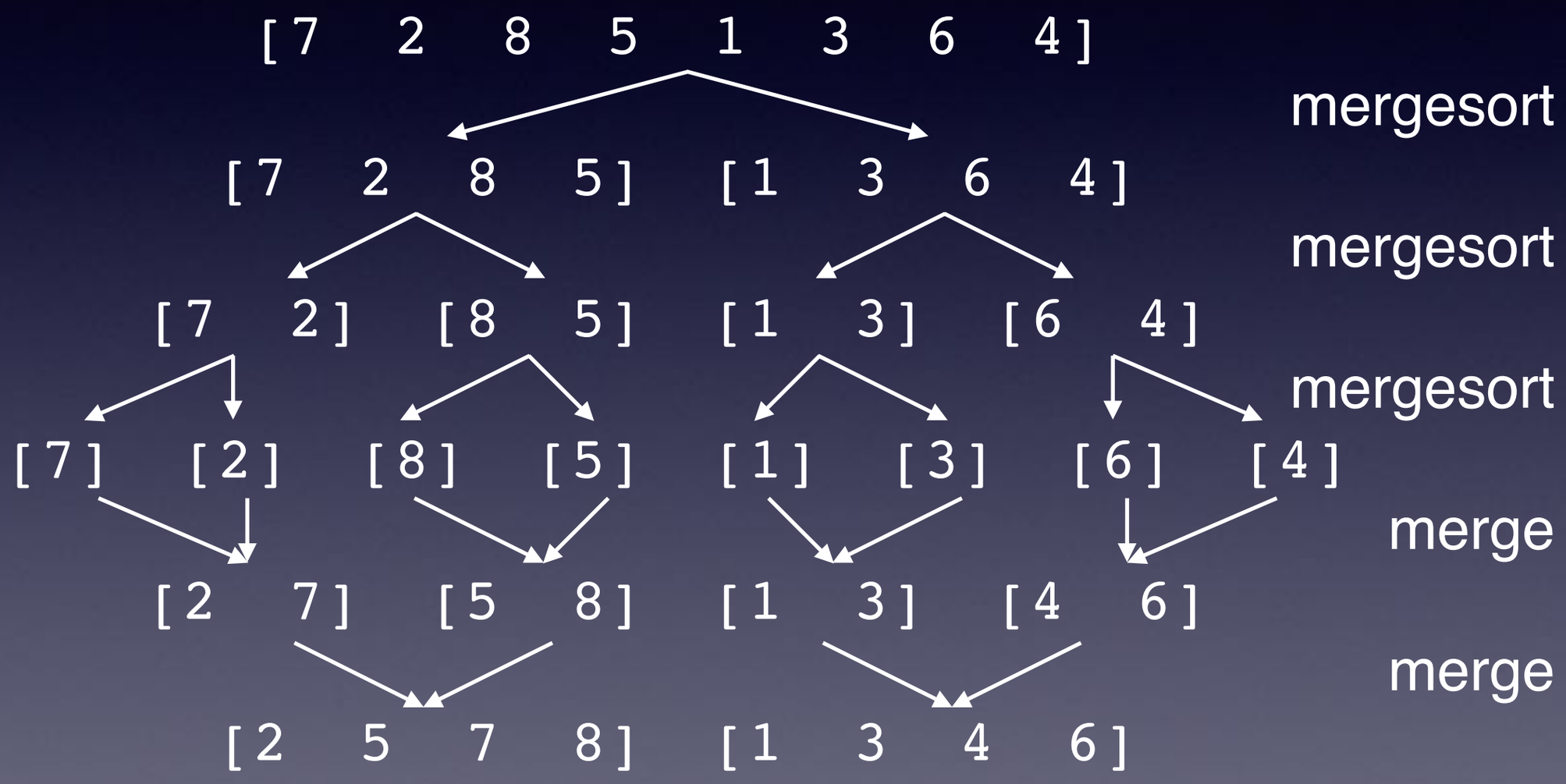
# Mergesort

Here's how these functions would sort a list of integers:



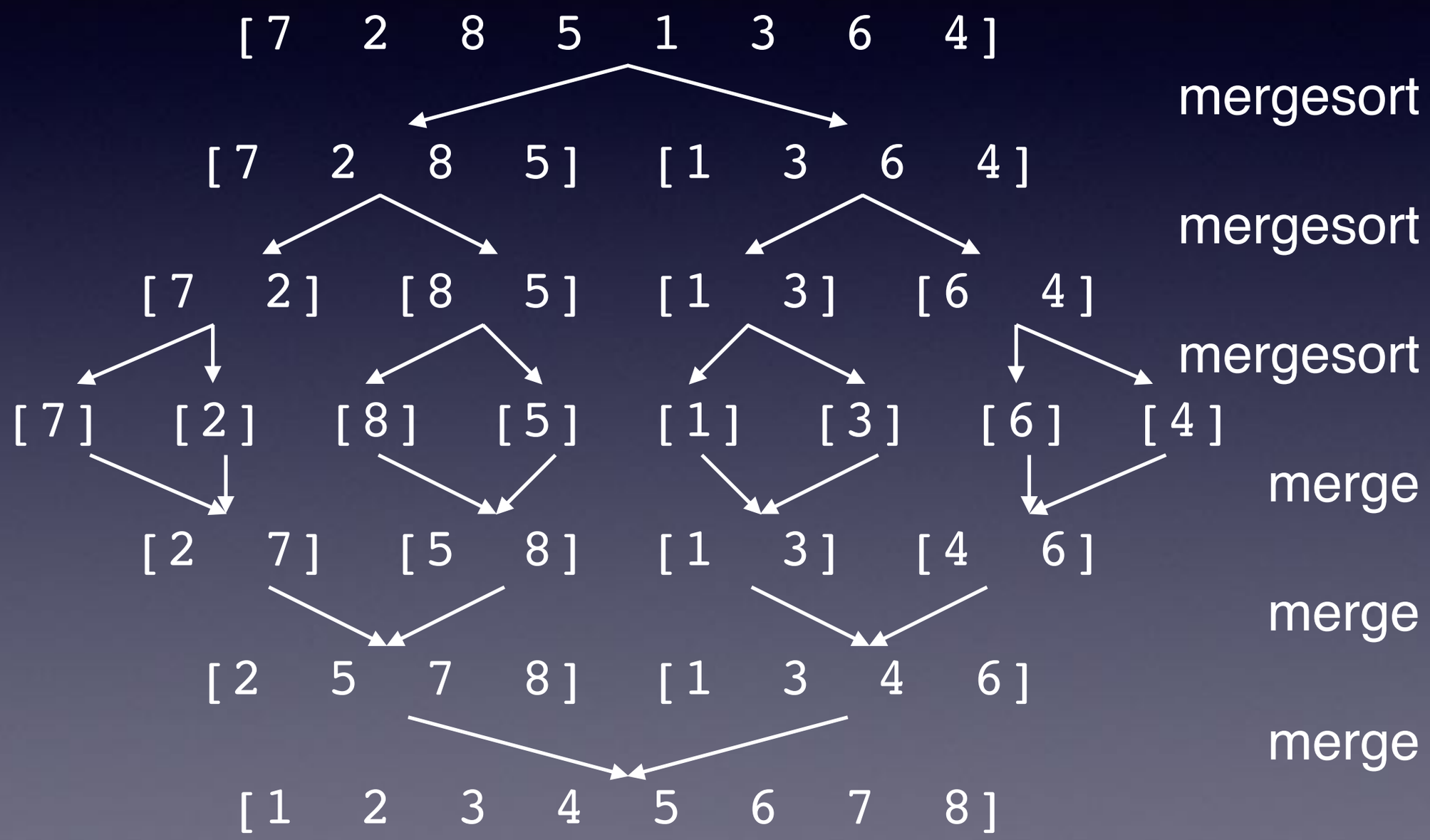
# Mergesort

Here's how these functions would sort a list of integers:



# Mergesort

Here's how these functions would sort a list of integers:



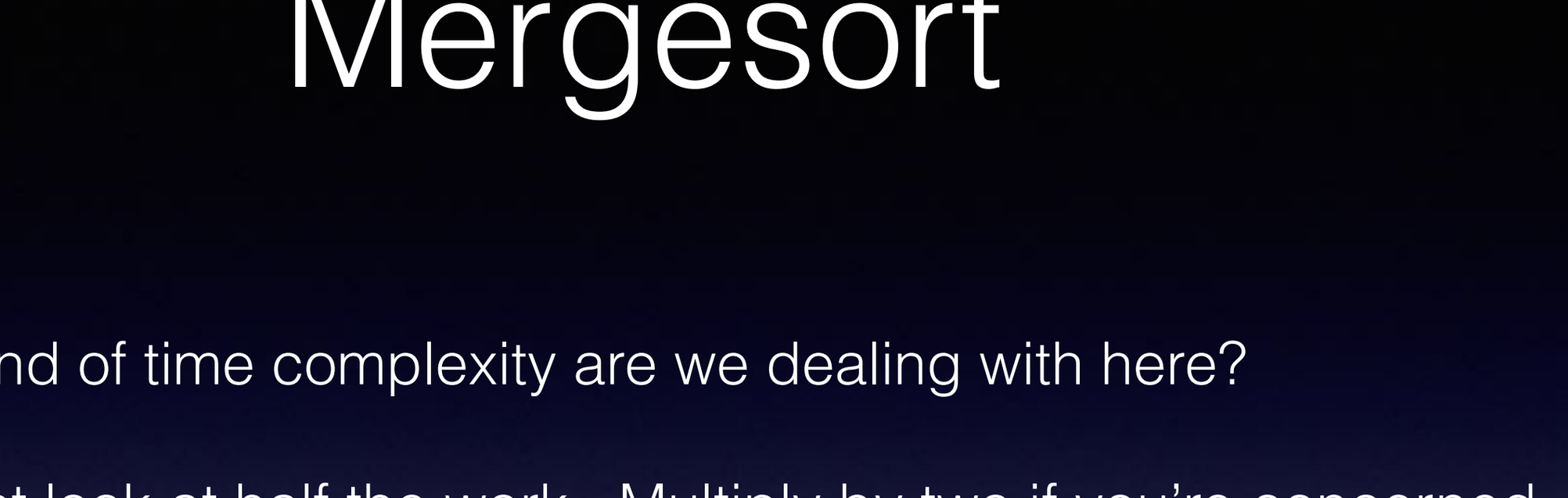
# Mergesort

What kind of time complexity are we dealing with here?

# Mergesort

What kind of time complexity are we dealing with here?

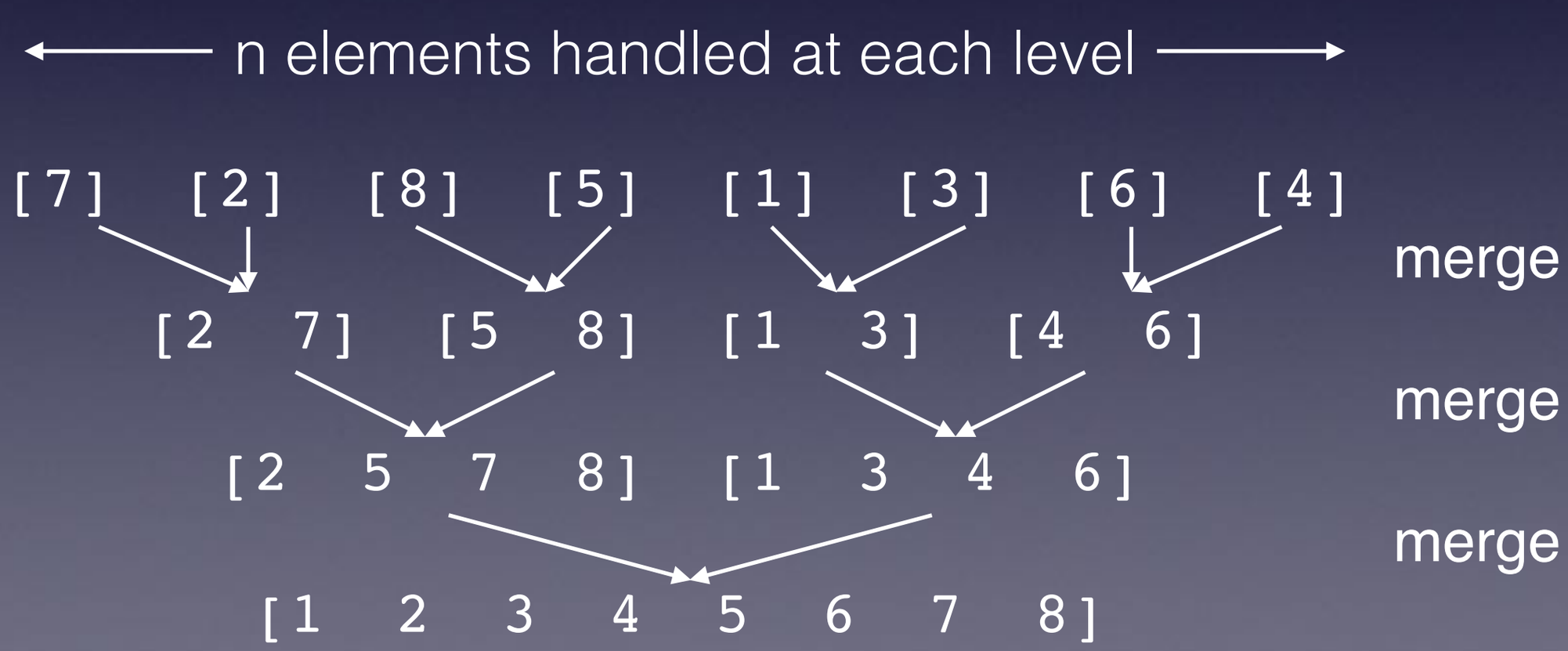
Let's just look at half the work.  Multiply by two if you're concerned.



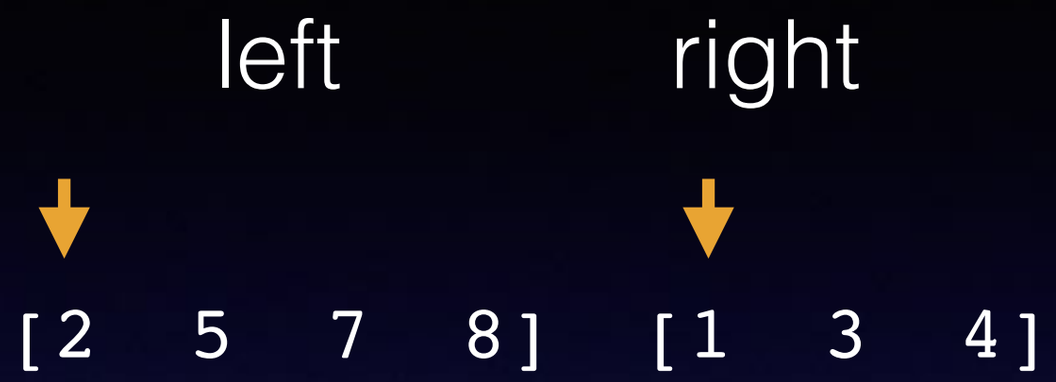
# Mergesort

Those last three lines represent the merge of n elements from the original list to larger and larger lists, along with the required comparisons to maintain sorted order while merging.

What's the complexity of each round of merging?



# Merging two sorted lists

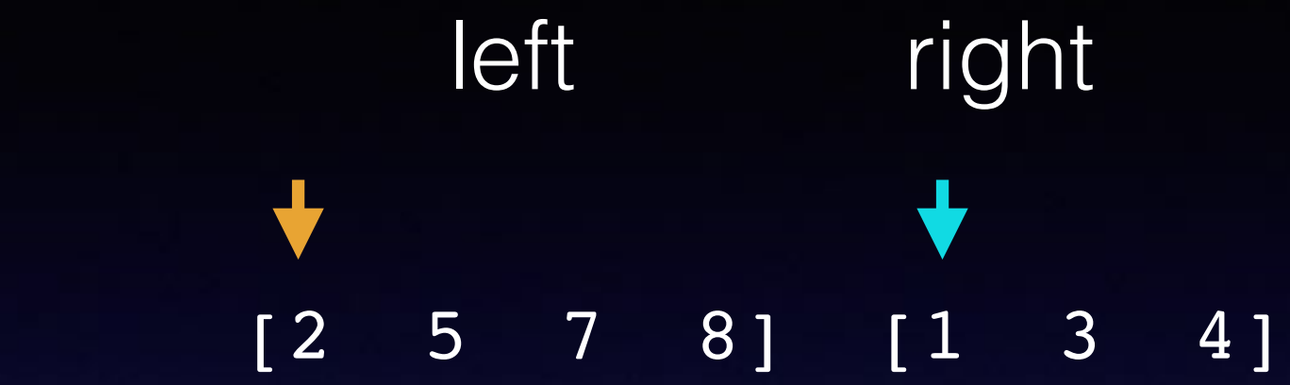



Here is an iterative algorithm for merging similar to how it is done in the book. The code is not as simple as the recursive algorithm on the previous slide, but it's useful for illustrating the complexity of merge.

# Merging two sorted lists


We have pointers to lowest unvisited elements in the **left** and **right** lists and a pointer to the next element to be inserted in the **sorted** list. We can always know the length of the sorted list if we need it.

# Merging two sorted lists



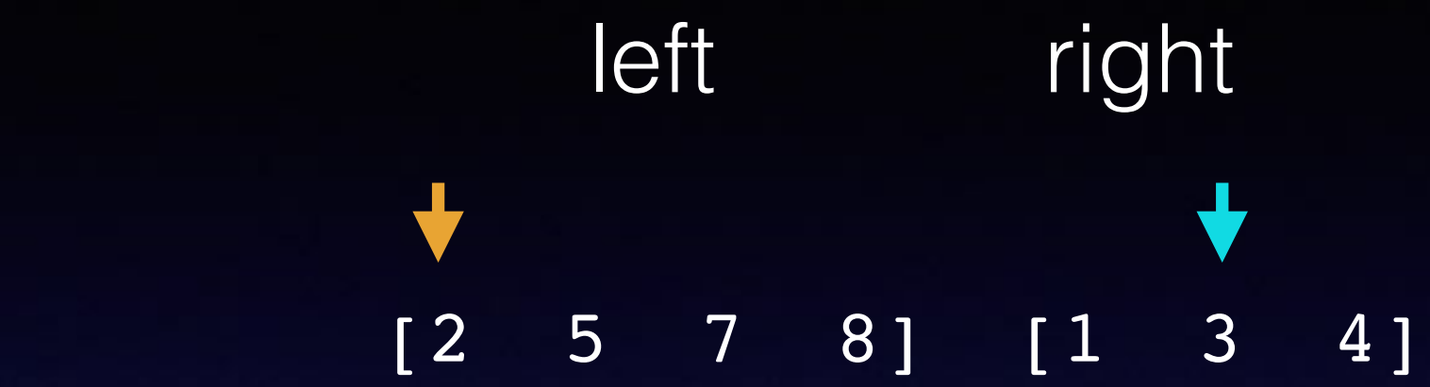

Do **left** and **right** both have elements remaining? Yes
Is 2 less than 1? No

# Merging two sorted lists


sorted


Do **left** and **right** both have elements remaining? Yes
Is 2 less than 1? No
Insert 1 from right list (the smaller element)
into next position of the sorted list

# Merging two sorted lists

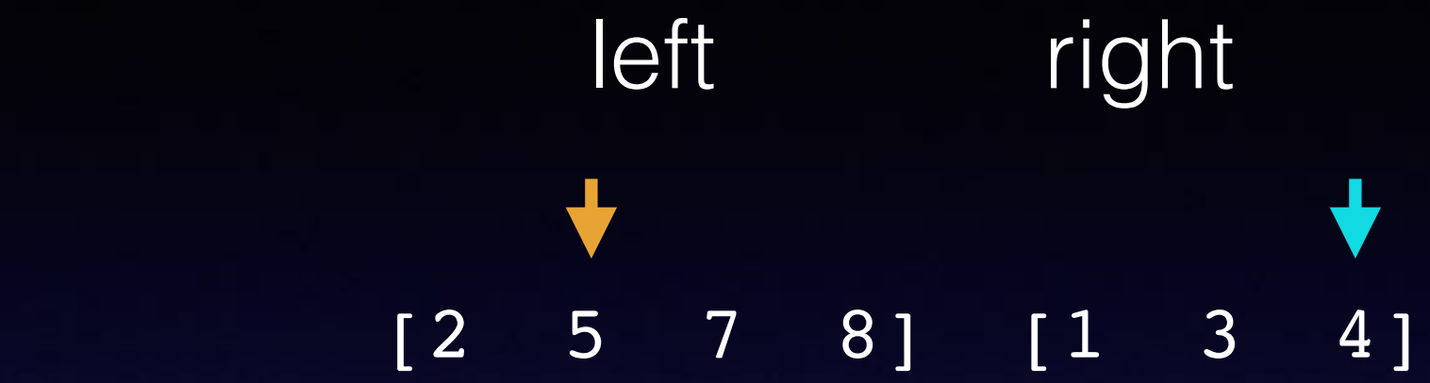



Do **left** and **right** both have elements remaining? Yes
Is 2 less than 3? Yes
Insert 2 from left list (the smaller element)
into next position of the sorted list

# Merging two sorted lists

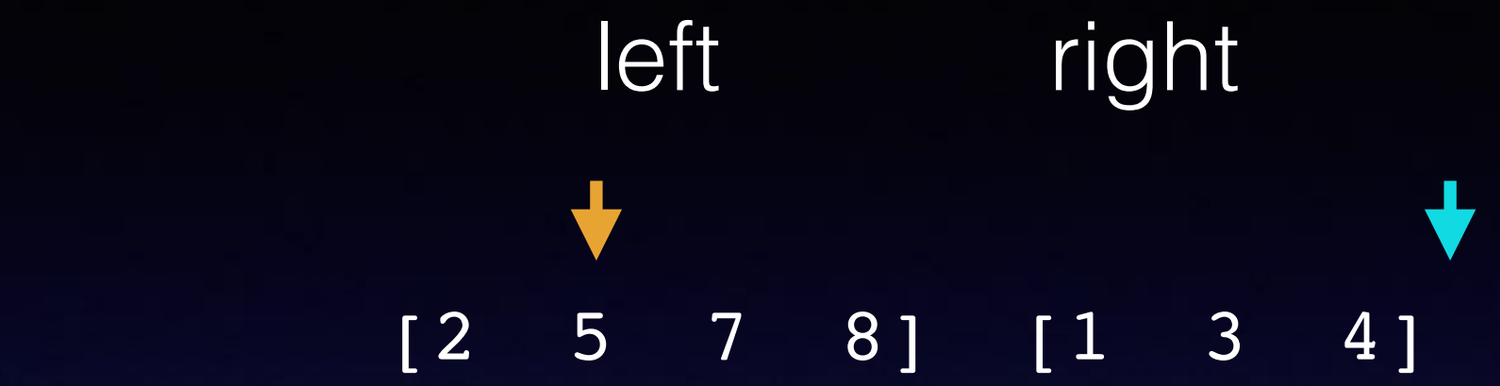

Do **left** and **right** both have elements remaining? Yes
Is 5 less than 3? No
Insert 3 from right list (the smaller element)
into next position of the sorted list

# Merging two sorted lists



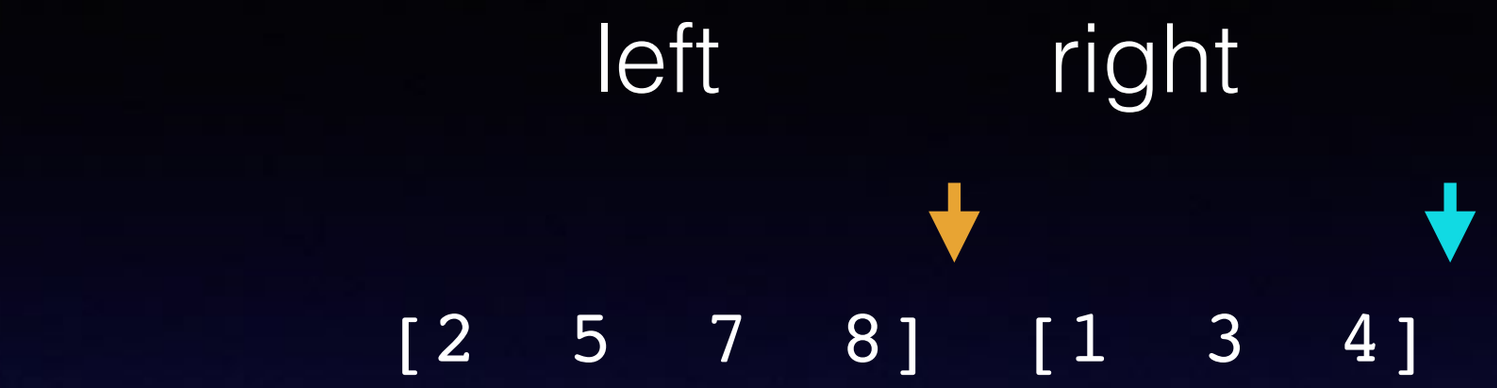

Do **left** and **right** both have elements remaining? Yes
Is 5 less than 4? No
Insert 4 from right list (the smaller element)
into next position of the sorted list

# Merging two sorted lists



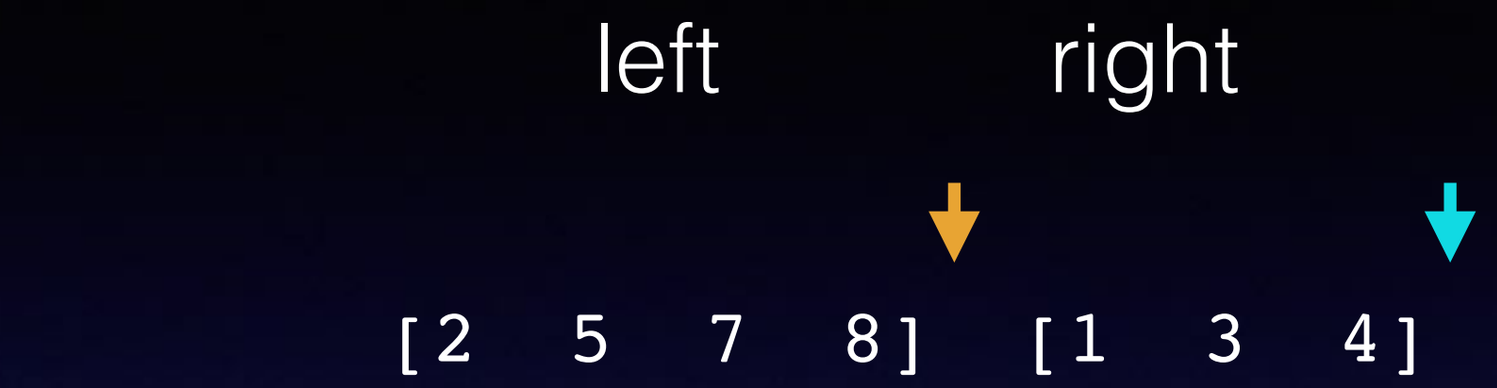

Do **left** and **right** both have elements remaining? **No**
It must be that the remaining elements of the sorted left list are all larger than the elemented in the sorted list.
So insert all remaining elements of **left** list into sorted list.

# Merging two sorted lists




What's the complexity?

# Merging two sorted lists

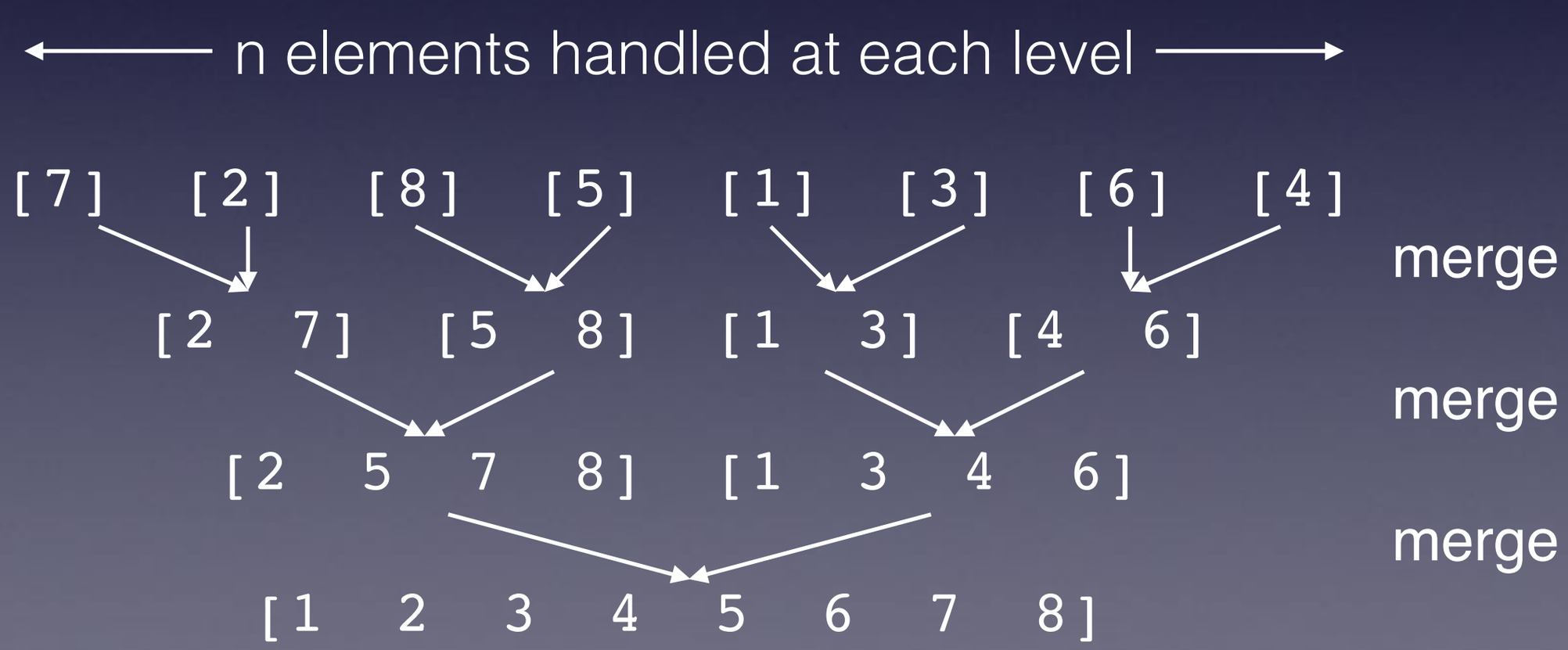



What's the complexity?

If k is combined length of the two input lists, or equivalently the sorted output list. Then the procedure is O(k) since we traversed each list only once and did a constant number of operations for each element in the output.

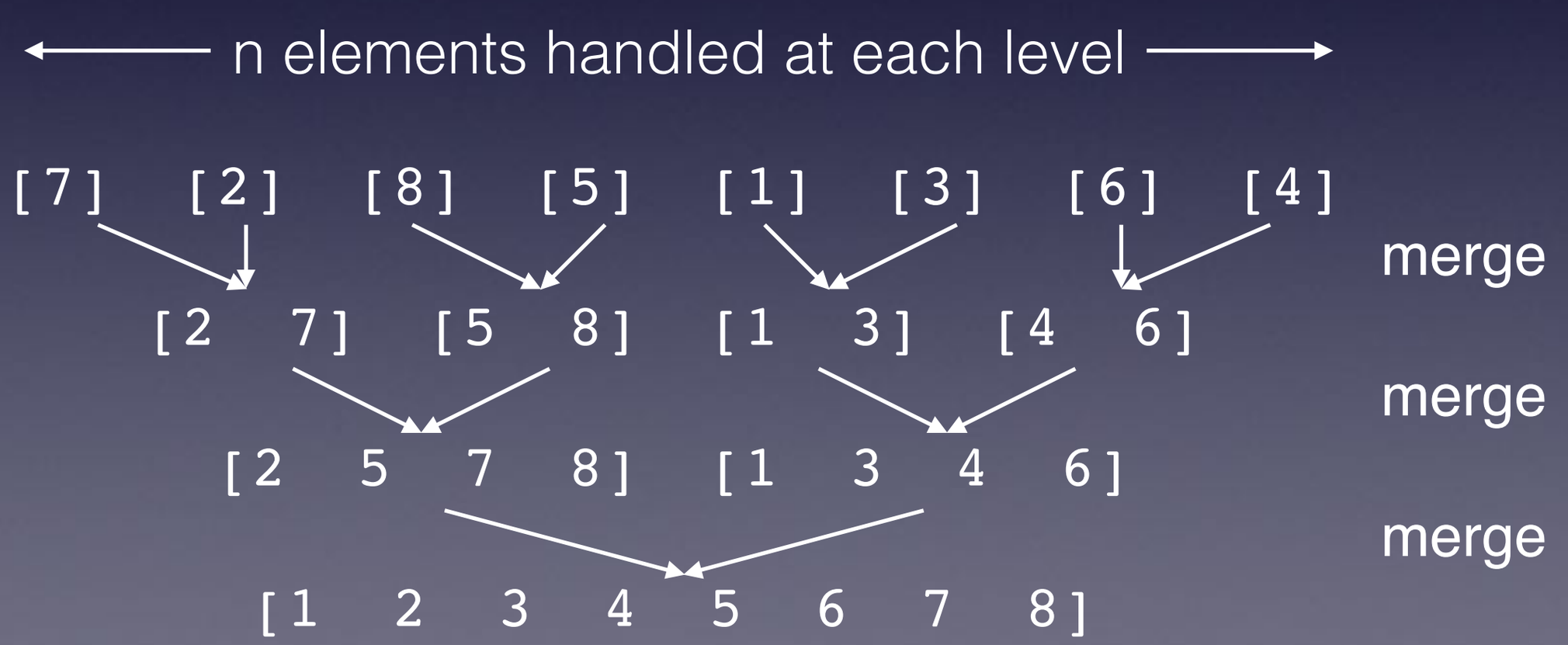
# Mergesort

Those last three lines represent the merge of n elements from the original list to larger and larger lists, along with the required comparisons to maintain sorted order while merging. **So each round of merging is O(n)**.



# Mergesort

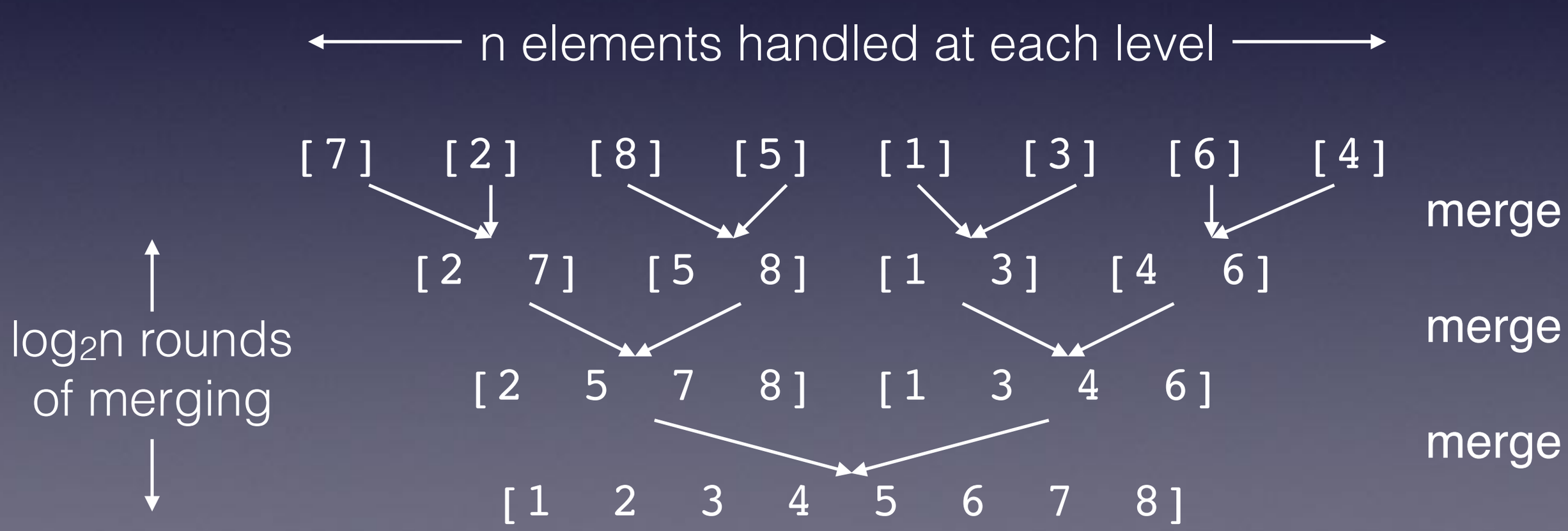
There are 3 rounds of merging when n = 8.  How many rounds of merging would be required if n = 16?  n = 32?  What does that tell you?



# Mergesort

There are 3 rounds of merging when n = 8.  How many rounds of merging would be required if n = 16?  n = 32?  What does that tell you?

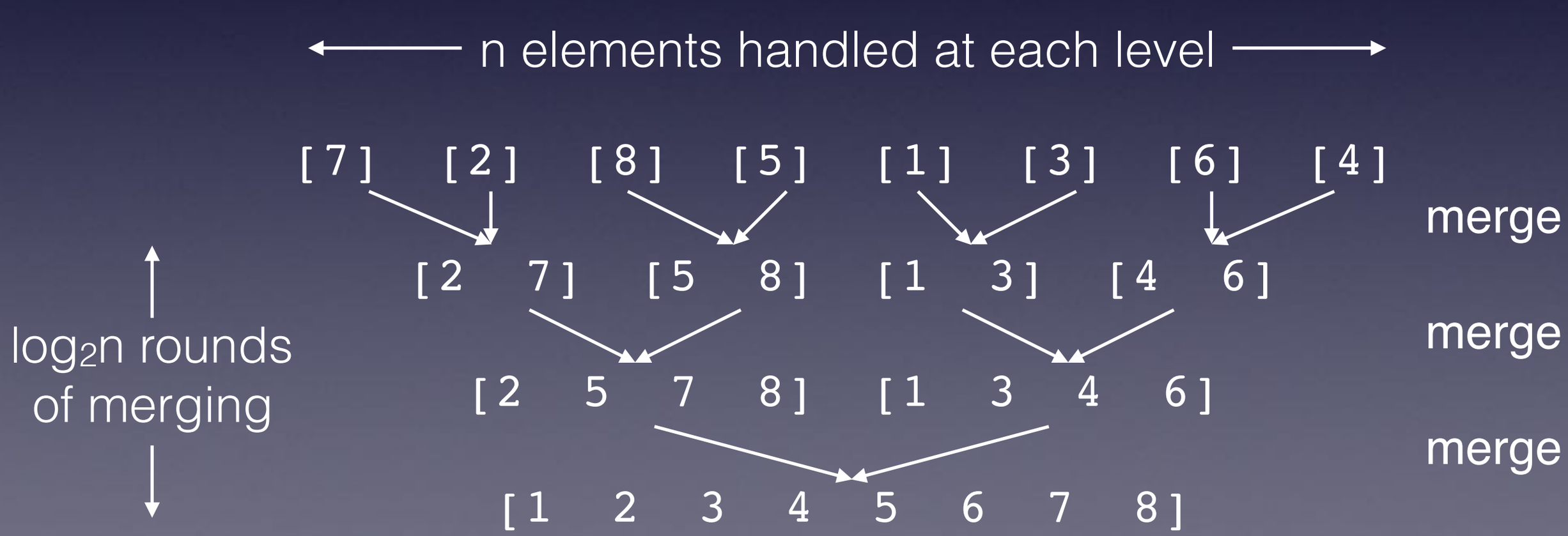
How does O(log n) sound?



# Mergesort

What kind of time complexity are we dealing with here?

Worst case time complexity for mergesort is $O(n \log_2 n)$.



# Mergesort

The Python functions below implement mergesort (this is not the textbook version, but may be easier to follow...).

```
def mergesort(mlist):
    if len(mlist) < 2:
        return mlist
    else:
        mid = len(mlist)//2
        return merge(mergesort(mlist[:mid]),mergesort(mlist[mid:]))

# merge two sorted lists
def merge(left, right):
    if left == []:
        return right
    elif right == []:
        return left
    elif left[0] < right[0]:
        return [left[0]] + merge(left[1:], right)
    else:
        return [right[0]] + merge(left, right[1:])
```

# Mergesort

Mergesort without making multiple copies of lists, i.e. slicing for each recursive call, gets a somewhat different implementation.

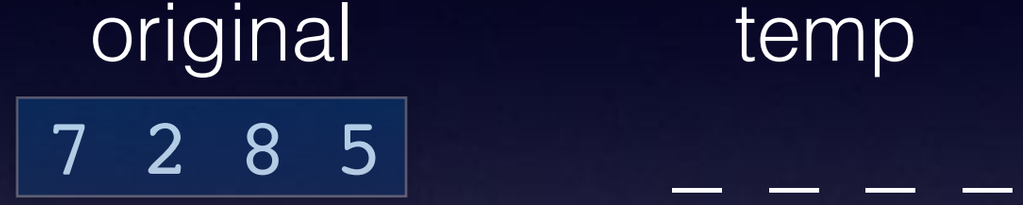
The typical implementation involves creating a temporary array to hold the results of merging two sorted subarrays (demarcated in the original array). Then that sorted result in the temporary array is copied back to the original array.

This approach only takes up 2 * n (i.e. O(n)) space.

Why the need for a temporary array? Try performing mergesort in place. The merge step really goobers things up.

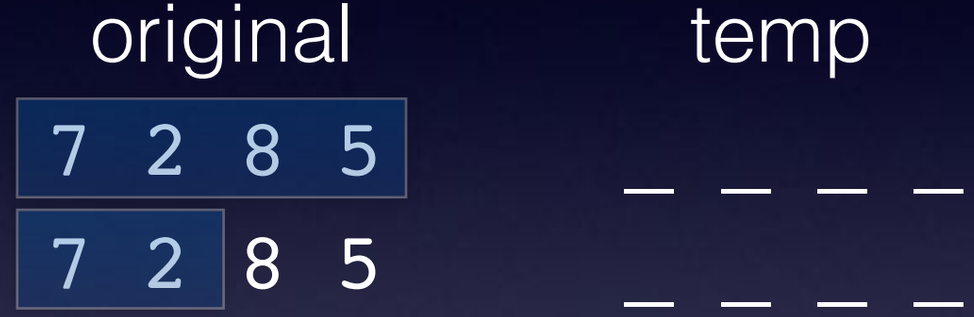
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



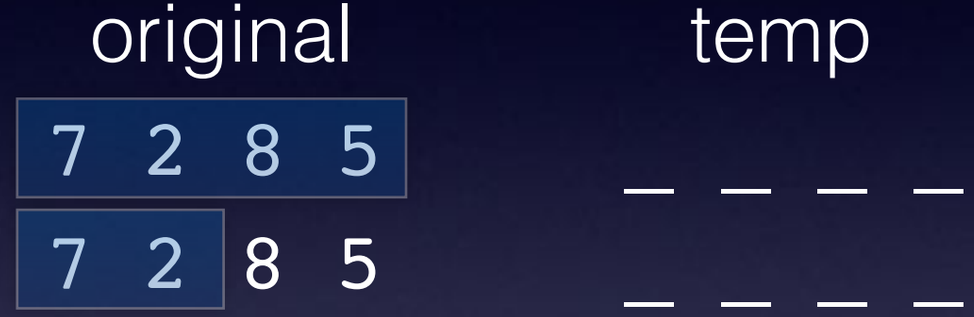
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



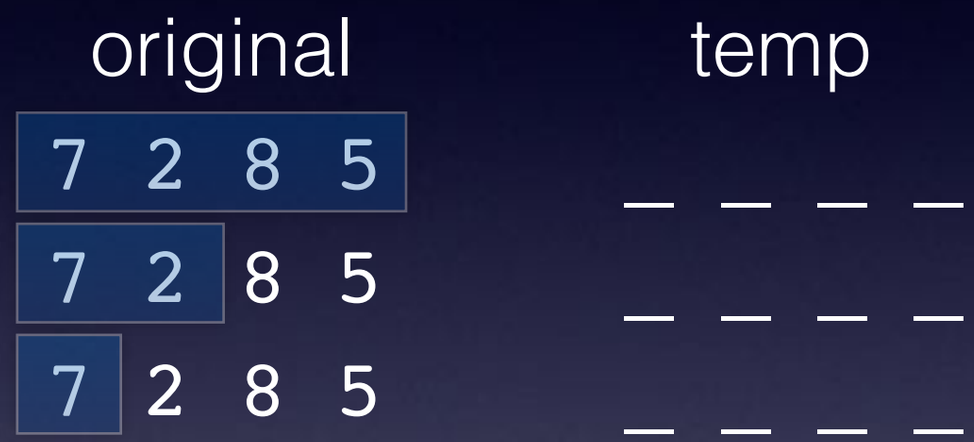
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



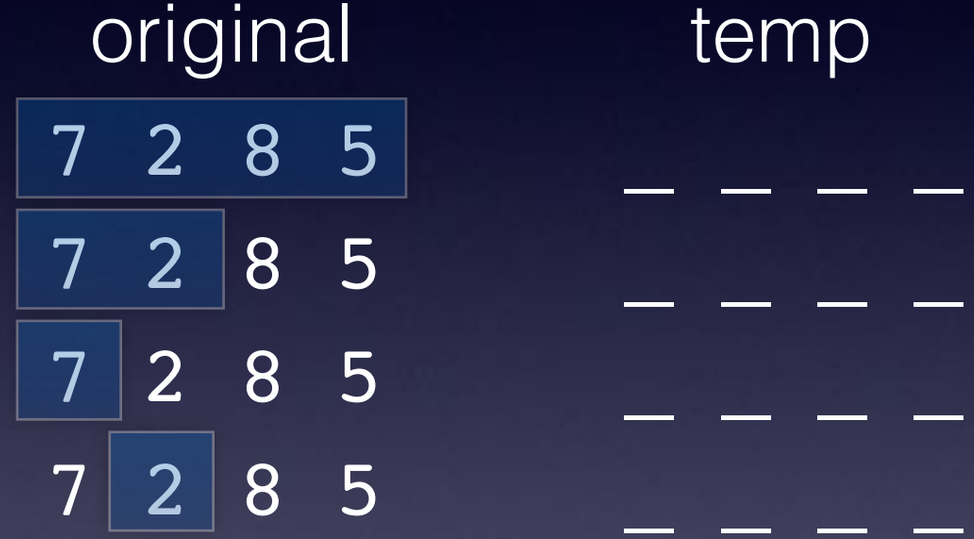
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



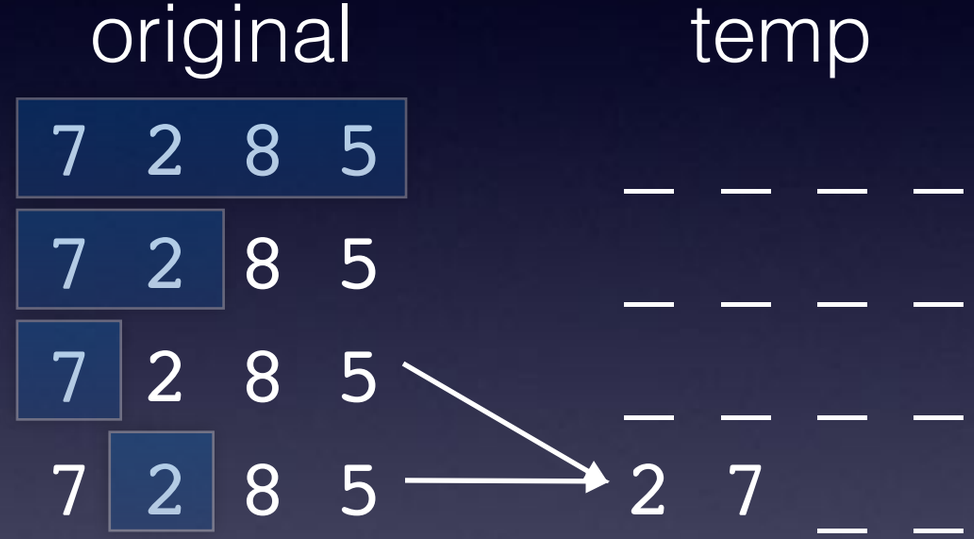
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



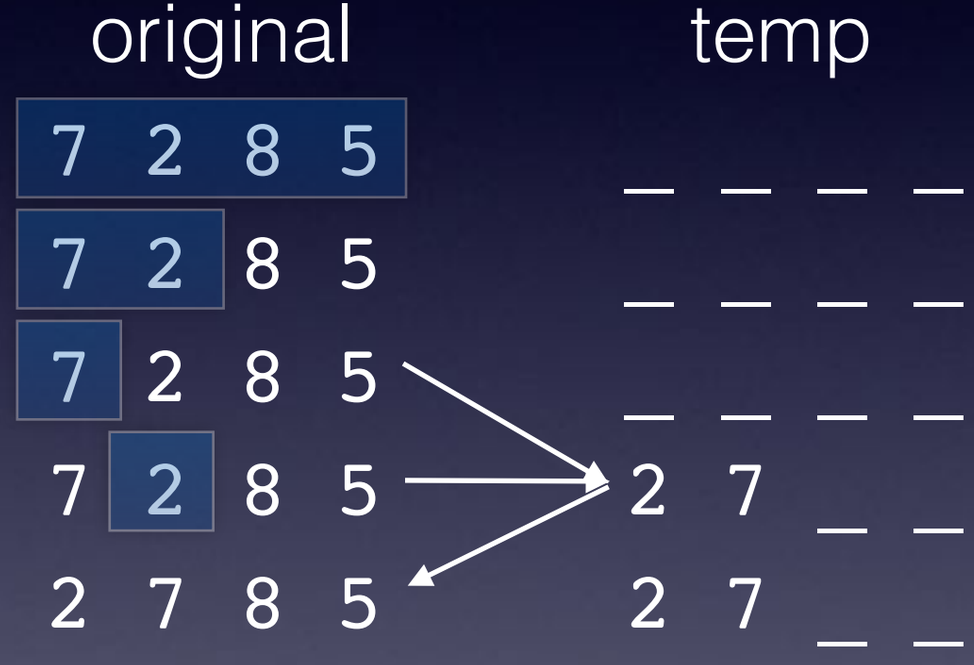
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



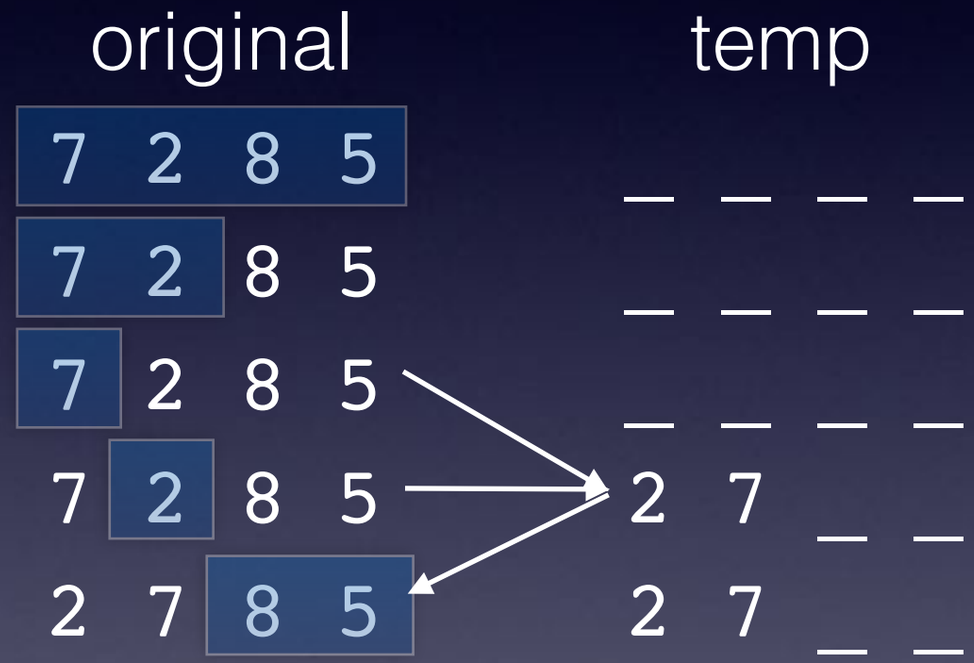
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



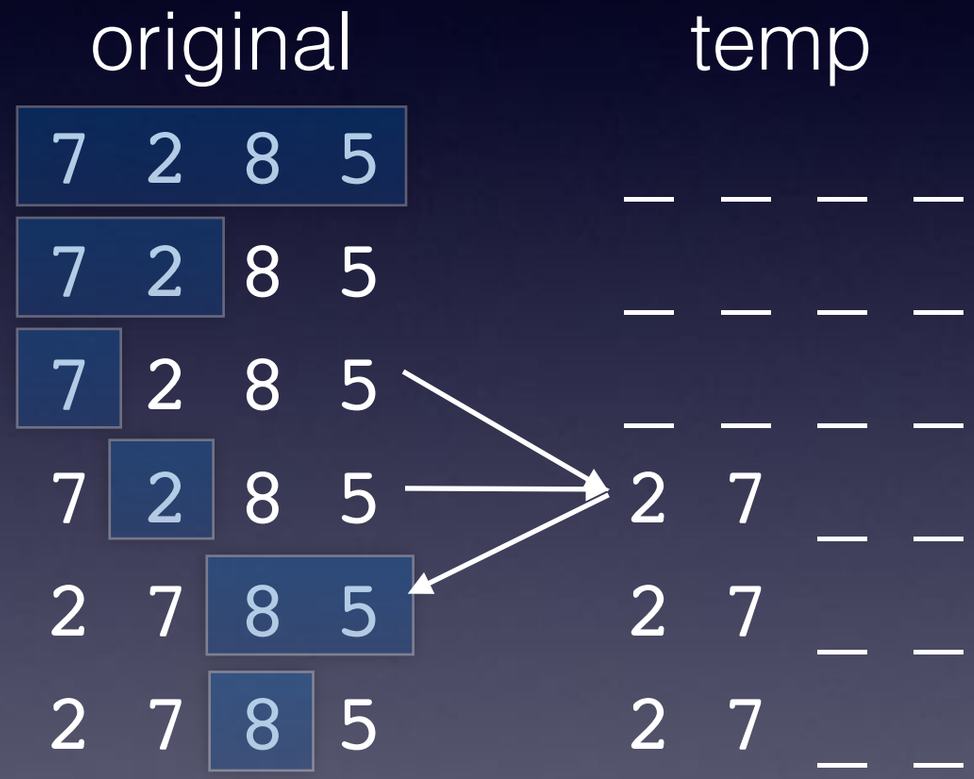
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



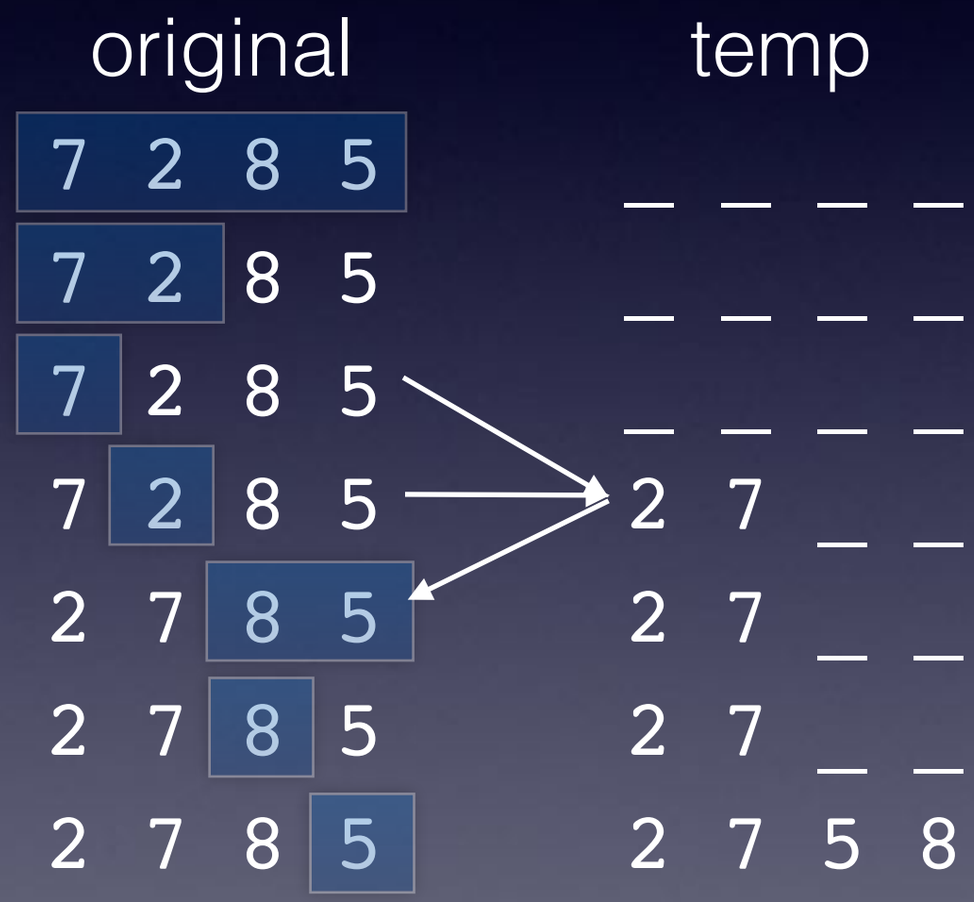
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



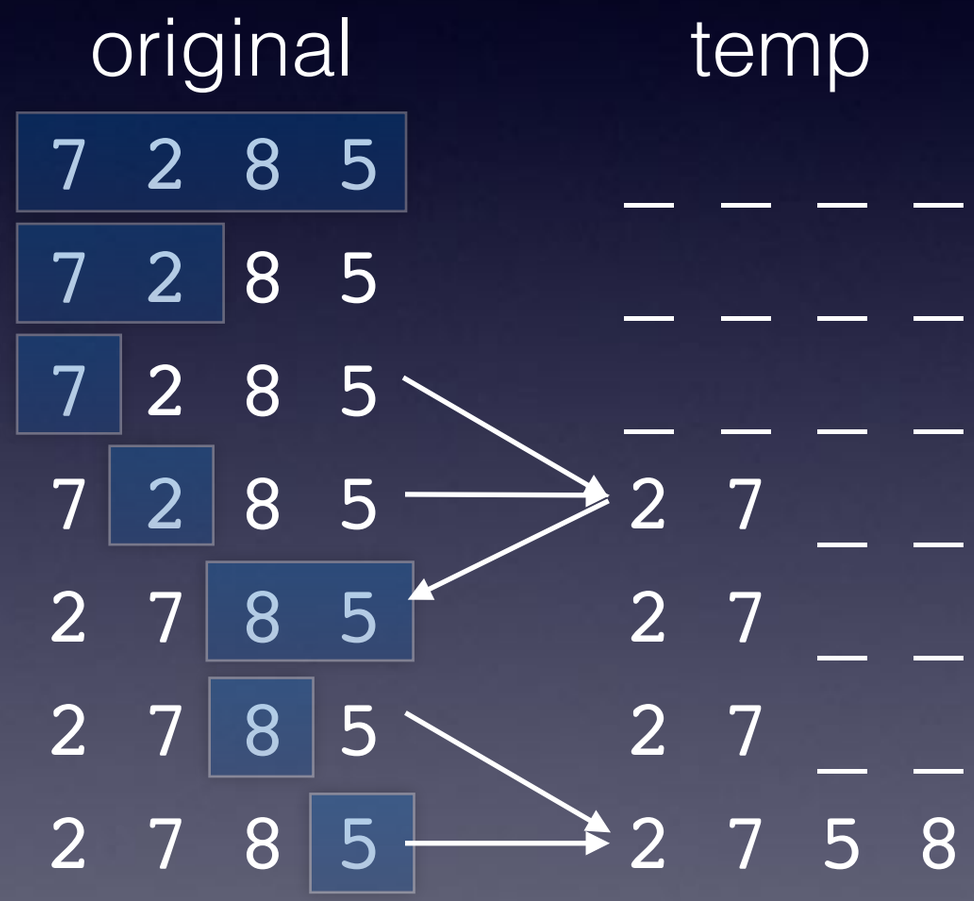
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



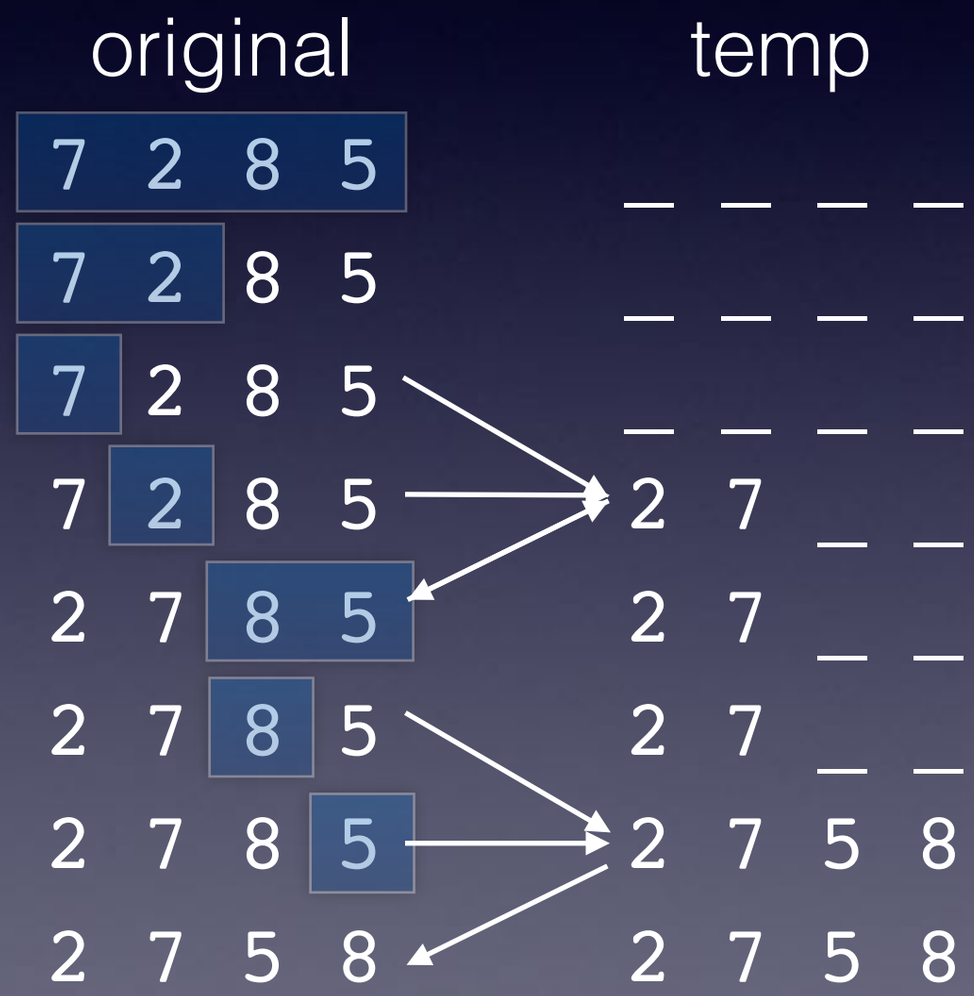
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



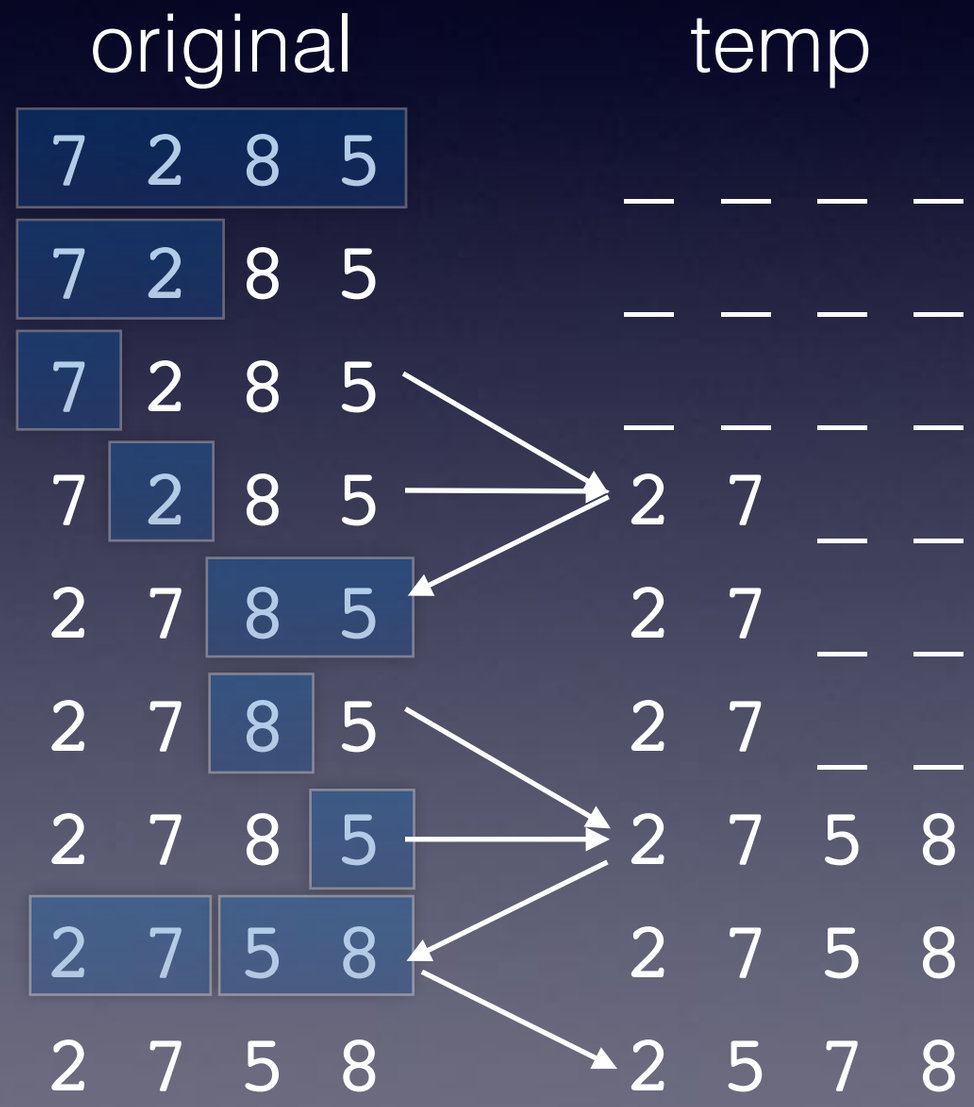
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



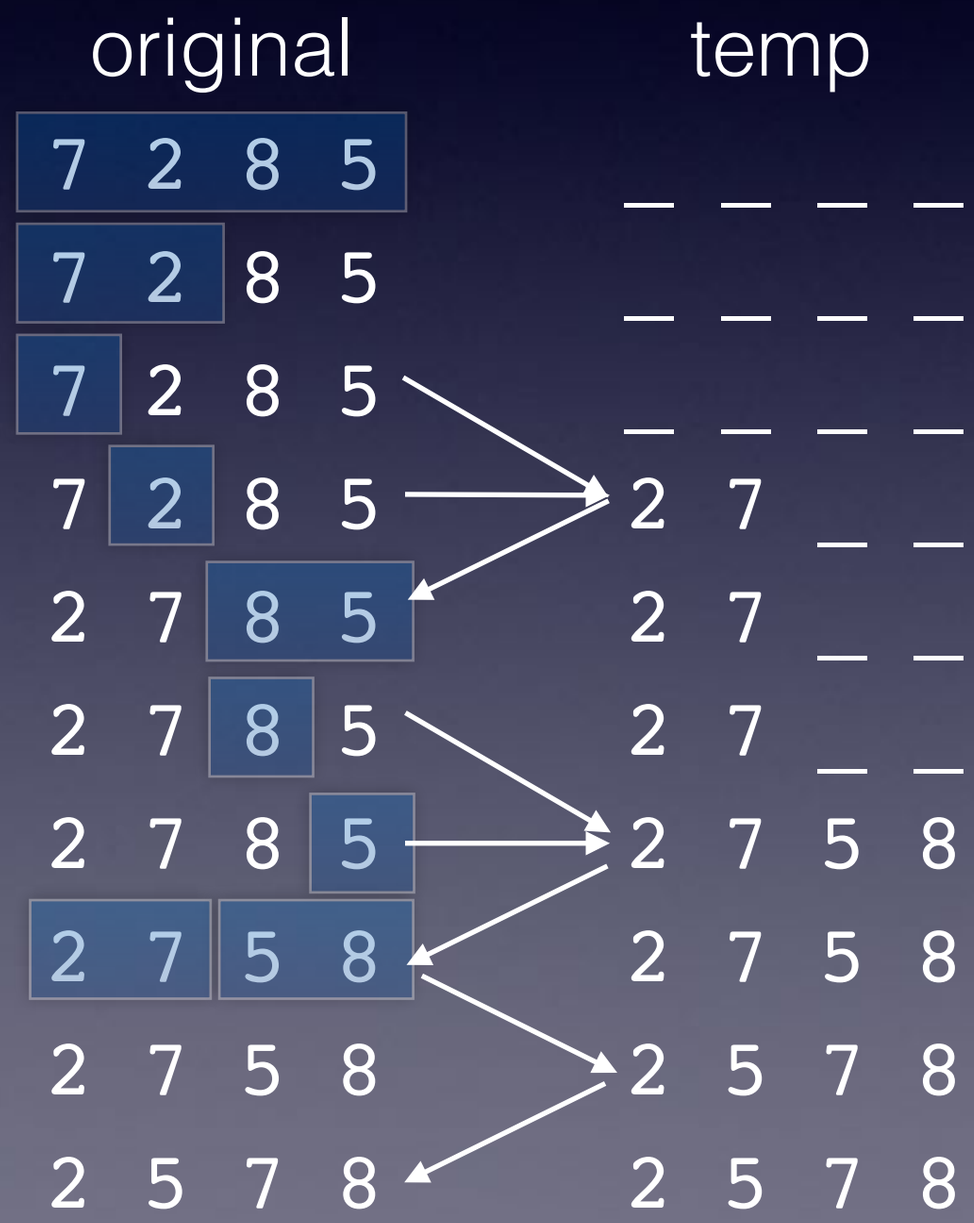
# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



# Mergesort

Here's an imperfect trace of merge sorting just four values in an array (assume we've drawn little boxes to represent the arrays:



# Mergesort

If mergesort can be done in 2n space,
then it can be O(n) space complexity