

# CS 106B

## Lecture 11: Sorting

Wednesday, April 26, 2017

---

Programming Abstractions  
Spring 2017  
Stanford University  
Computer Science Department

Lecturer: Chris Gregg

reading:  
Programming Abstractions in C++, Section 10.2



# Today's Topics

- Logistics
  - Practice Midterm: Thursday 8-10pm, Bishop Auditorium.
  - Real Midterm: Next **Thursday**, 7-9pm. Please email if you need an alternate time!
  - Tiny Feedback:
    - *Maybe post the Tiny Feedback to the website (don't waste time)*
- Sorting
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quicksort
  - Other sorts you might want to look at:
    - Radix Sort
    - Shell Sort
    - Tim Sort
    - Heap Sort (we will cover heaps later in the course)
    - Bogosort



# Sorting!

- In general, sorting consists of putting elements into a particular order, most often the order is numerical or lexicographical (i.e., alphabetic).
- In order for a list to be sorted, it must:
  - be in nondecreasing order (each element must be no smaller than the previous element)
  - be a permutation of the input



# Sorting!

- Sorting is a well-researched subject, although new algorithms do arise (see Timsort, from 2002)
- Fundamentally, *comparison* sorts at best have a complexity of  **$O(n \log n)$** .
- We also need to consider the space complexity: some sorts can be done in place, meaning the sorting does not take extra memory. This can be an important factor when choosing a sorting algorithm!



(must sort)



# Sorting!

- In-place sorting can be “stable” or “unstable”: a stable sort retains the order of elements with the same key, from the original unsorted list to the final, sorted, list
- There are some phenomenal online sorting demonstrations: see the “Sorting Algorithm Animations” website:



- <http://www.sorting-algorithms.com>, or the animation site at: <http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html> or the cool “15 sorts in 6 minutes” video on YouTube: <https://www.youtube.com/watch?v=kPRA0W1kECg>



# Sorts

- There are many, many different ways to sort elements in a list.  
We will look at the following:

Insertion Sort  
Selection Sort  
Merge Sort  
Quicksort



# Sorts

Insertion Sort

Selection Sort

Merge Sort

Quicksort



# Insertion Sort

Insertion sort: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

More specifically:

- consider the first item to be a sorted sublist of length 1
- insert second item into sorted sublist, shifting first item if needed
- insert third item into sorted sublist, shifting items 1-2 as needed
- ...
- repeat until all values have been inserted into their proper positions





# Insertion Sort

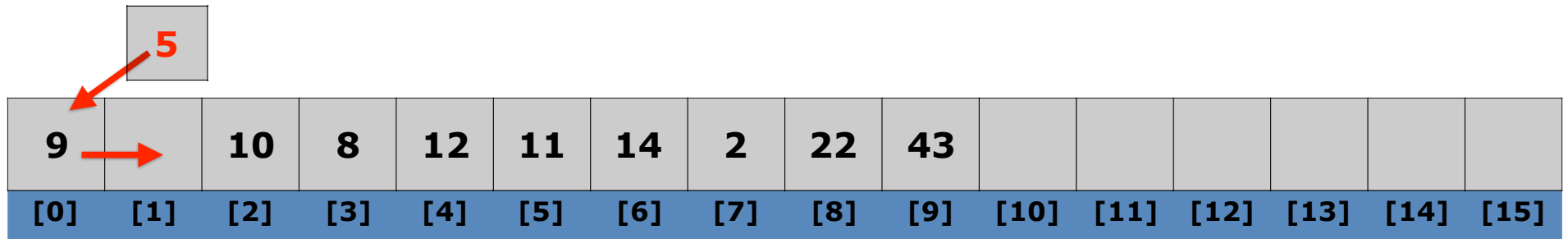
9	5	10	8	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort



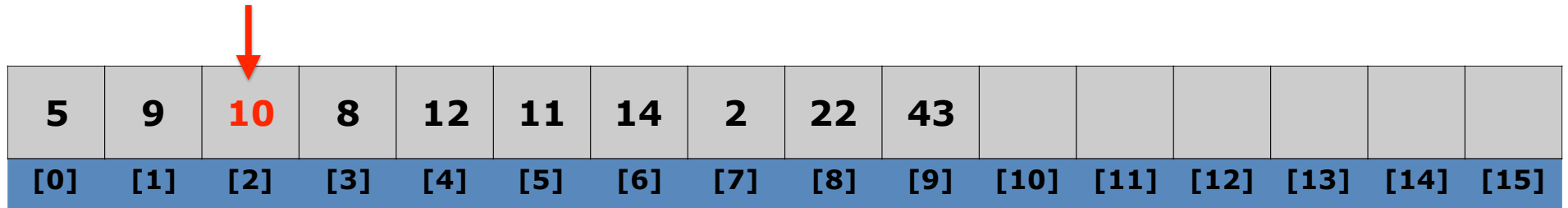
Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort

in place already (i.e., already bigger than 9)



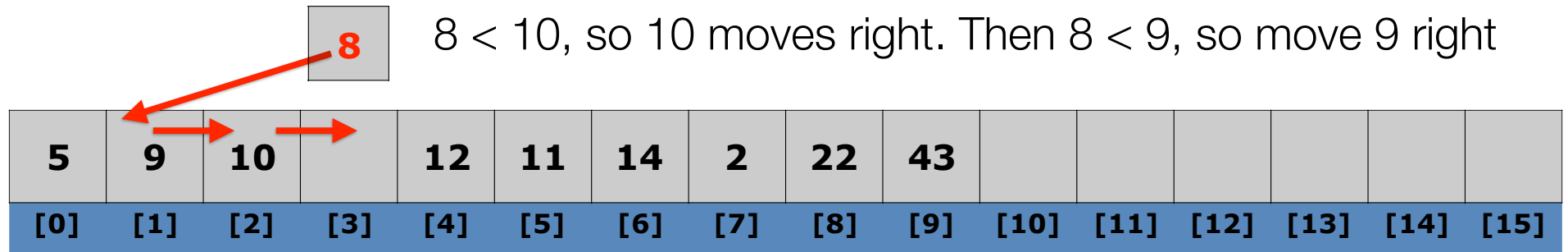
5	9	10	8	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort



Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort

in place already (i.e., already bigger than 10)

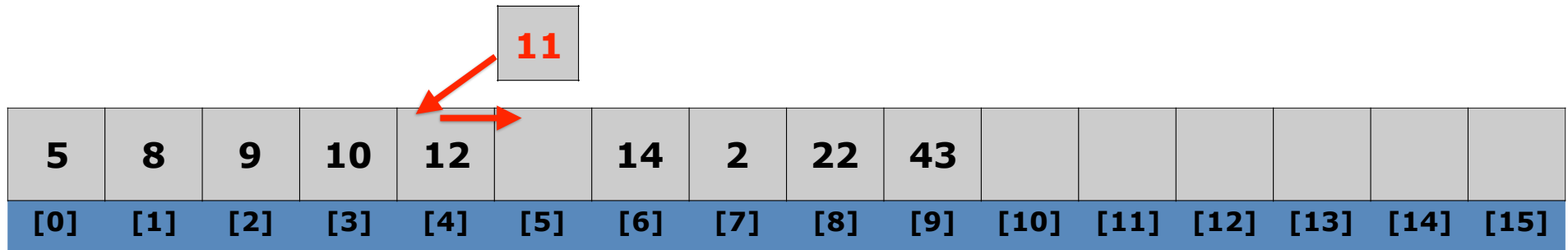
5	8	9	10	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort



Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort

in place already (i.e., already bigger than 12)

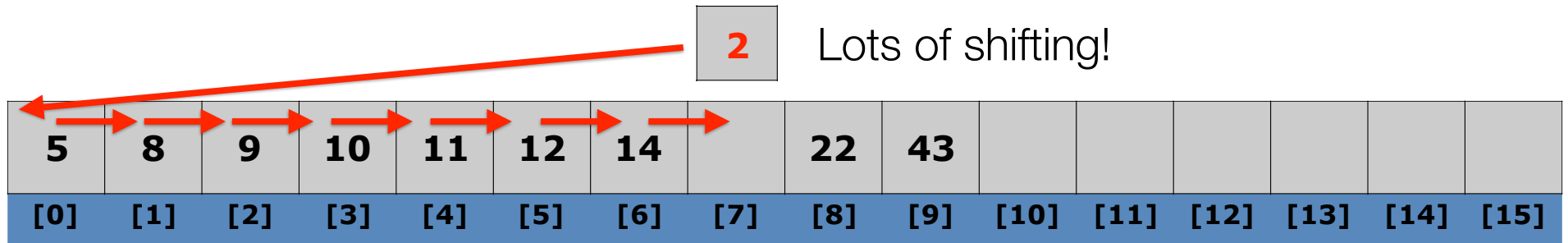
5	8	9	10	11	12	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort



Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.





# Insertion Sort

Okay



2	5	8	9	10	11	12	14	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Algorithm:

- iterate through the list (starting with the second element)
- at each element, shuffle the neighbors below that element up until the proper place is found for the element, and place it there.



# Insertion Sort

Okay



2	5	8	9	10	11	12	14	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

Complexity:

Worst performance:  $O(n^2)$  (why? -- see extra slide!)

Best performance:  $O(n)$

–Average performance:  $O(n^2)$  (but very fast for small arrays!)

–Worst case space complexity:  $O(n)$  total (plus one for swapping)



# Insertion Sort Code

```
// Rearranges the elements of v into sorted order.
void insertionSort(Vector<int>& v) {
    for (int i = 1; i < v.size(); i++) {
        int temp = v[i];
        // slide elements right to make room for v[i]
        int j = i;
        while (j >= 1 && v[j - 1] > temp) {
            v[j] = v[j - 1];
            j--;
        }
        v[j] = temp;
    }
}
```



# Sorts

Insertion Sort  
Selection Sort  
Merge Sort  
Quicksort



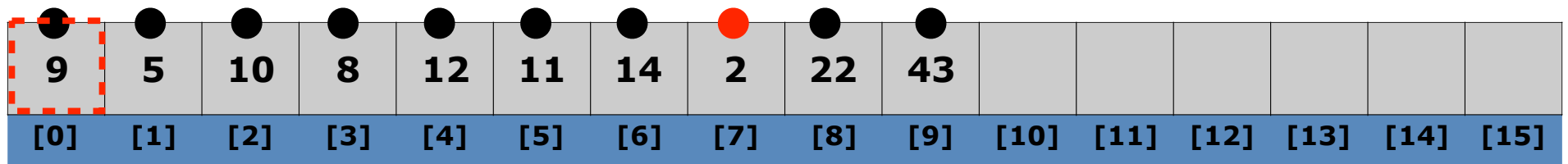
# Selection Sort

9	5	10	8	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- Selection Sort is another in-place sort that has a simple algorithm:
  - Find the smallest item in the list, and exchange it with the left-most unsorted element.
  - Repeat the process from the first unsorted element.
- See animation at: <http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>



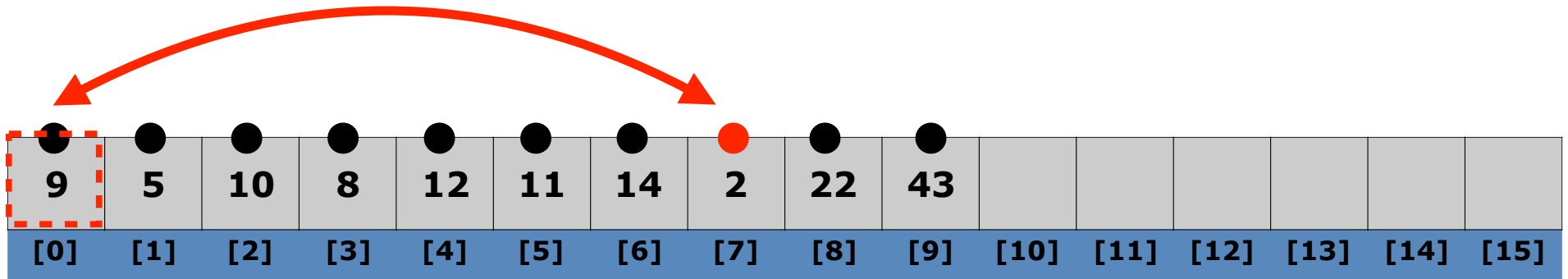
# Selection Sort



- Algorithm
  - Find the **smallest item in the list**, and exchange it with the left-most unsorted element.
  - Repeat the process from the first unsorted element.
- Selection sort is particularly slow, because it needs to go through **the entire list** each time to find the smallest item.



# Selection Sort

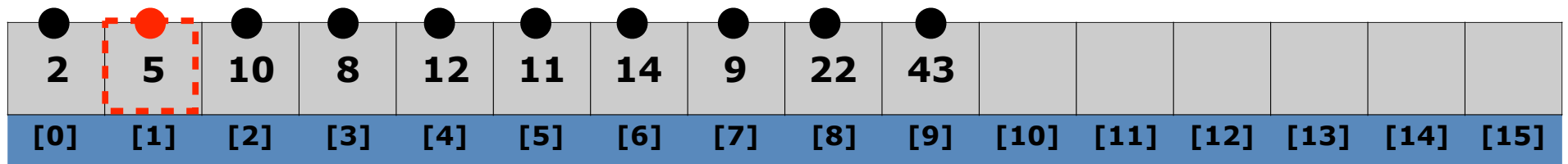


- Algorithm
  - Find the **smallest item in the list**, and exchange it with the left-most unsorted element.
  - Repeat the process from the first unsorted element.
- Selection sort is particularly slow, because it needs to go through **the entire list** each time to find the smallest item.



# Selection Sort

(no swap necessary)

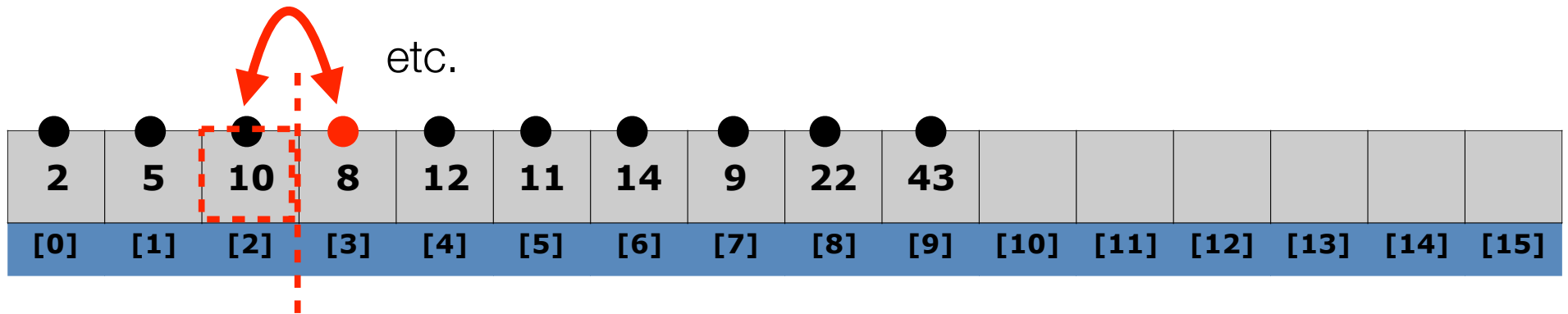


- Algorithm
  - Find the **smallest item in the list**, and exchange it with the left-most unsorted element.
  - Repeat the process from the first unsorted element.
- Selection sort is particularly slow, because it needs to go through **the entire list** each time to find the smallest item.





# Selection Sort



- Complexity:
  - Worst performance:  $O(n^2)$
  - Best performance:  $O(n^2)$
  - Average performance:  $O(n^2)$
  - Worst case space complexity:  $O(n)$  total (plus one for swapping)



# Selection Sort Code

```
// Rearranges elements of v into sorted order
// using selection sort algorithm
void selectionSort(Vector<int>& v) {
    for (int i = 0; i < v.size() - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < v.size(); j++) {
            if (v[j] < v[min]) {
                min = j;
            }
        }
        // swap smallest value to proper place, v[i]
        if (i != min) {
            int temp = v[i];
            v[i] = v[min];
            v[min] = temp;
        }
    }
}
```



# Sorts

Insertion Sort  
Selection Sort  
Merge Sort  
Quicksort



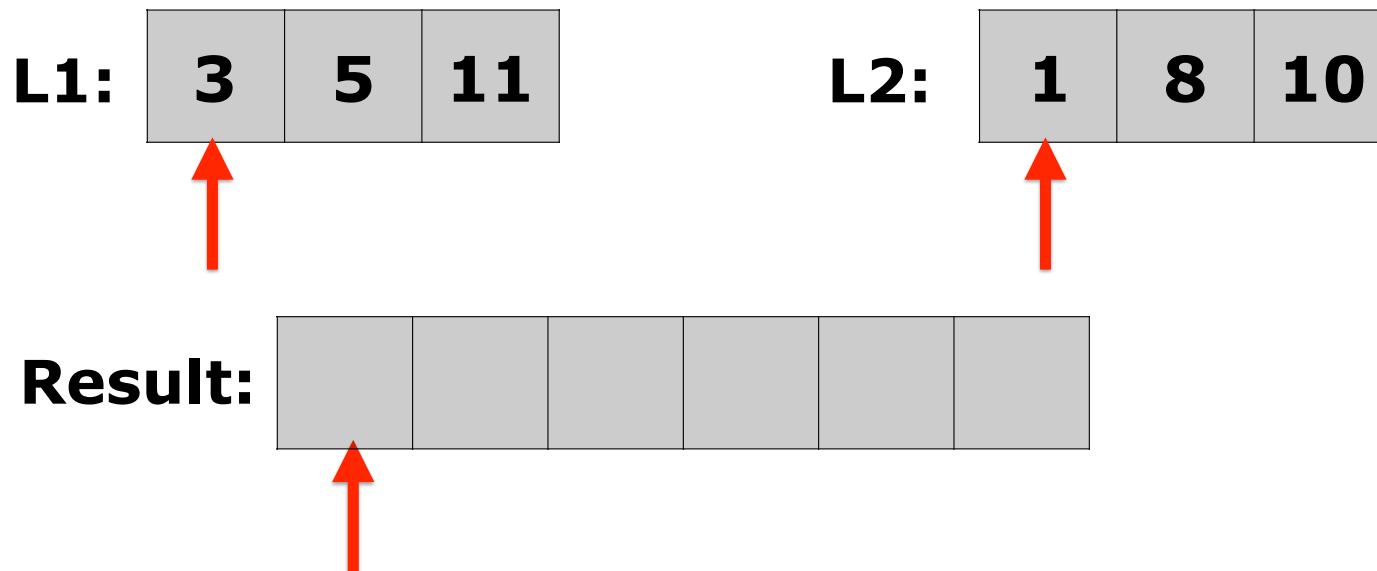
# Merge Sort

- Merge Sort is another comparison-based sorting algorithm and it is a *divide-and-conquer* sort.
- Merge Sort can be coded recursively
- In essence, you are merging sorted lists, e.g.,
- $L1 = \{3,5,11\}$     $L2 = \{1,8,10\}$
- $\text{merge}(L1,L2)=\{1,3,5,8,10,11\}$



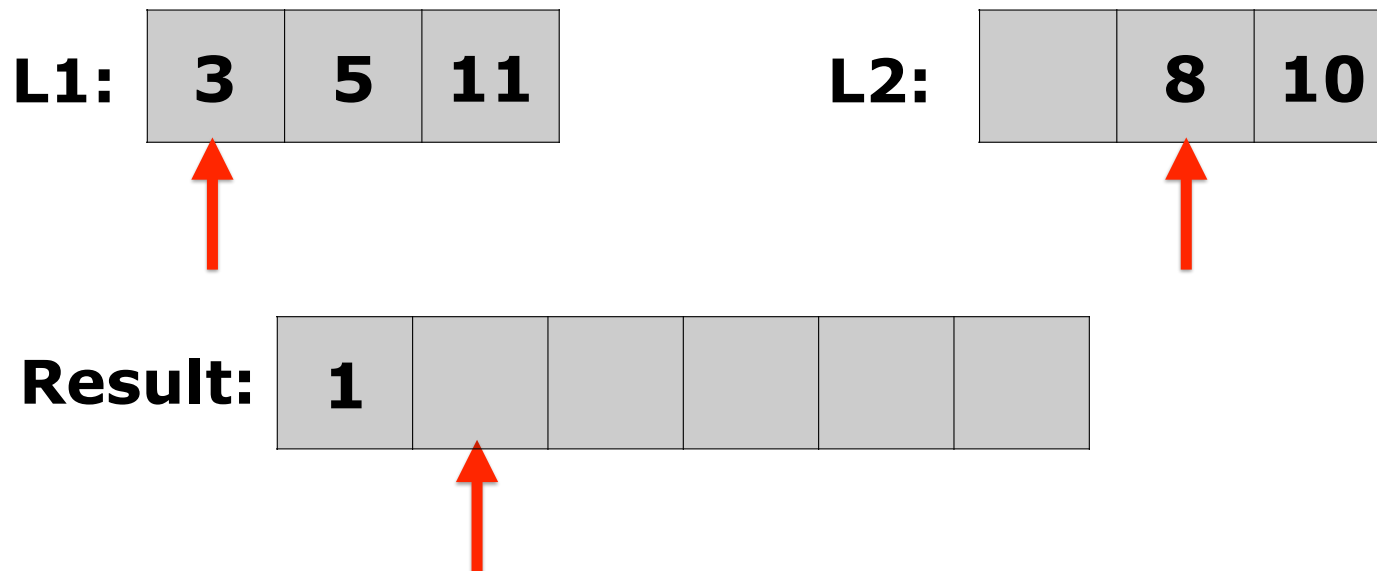
# Merge Sort

- Merging two sorted lists is easy:



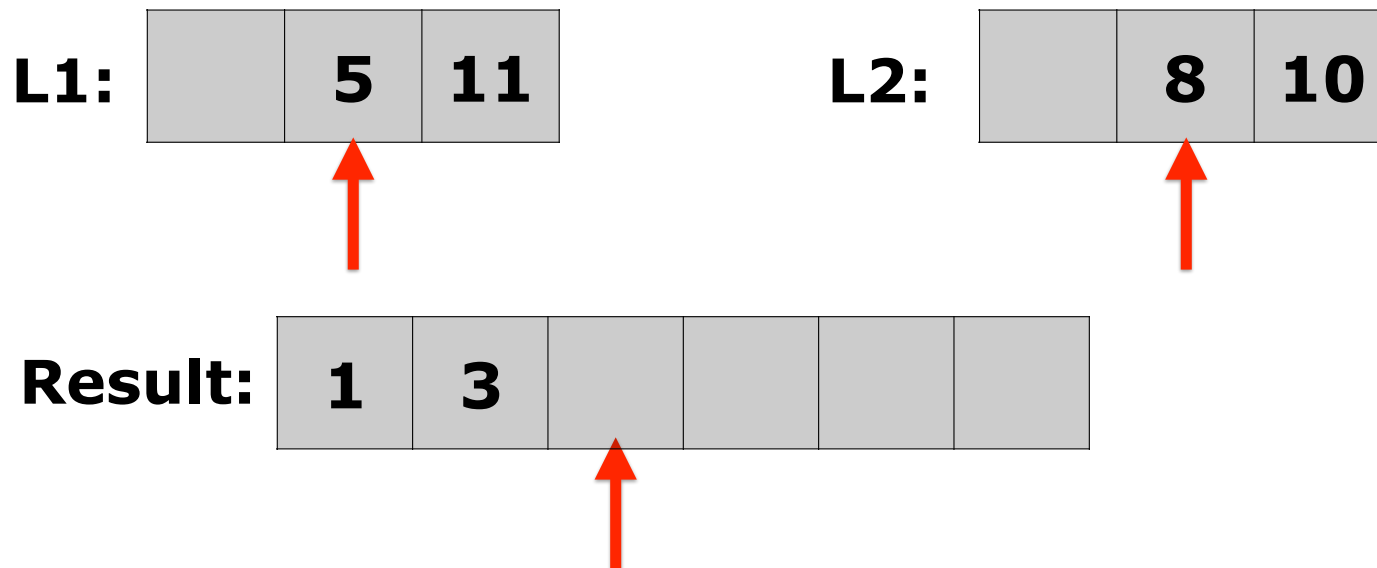
# Merge Sort

- Merging two sorted lists is easy:



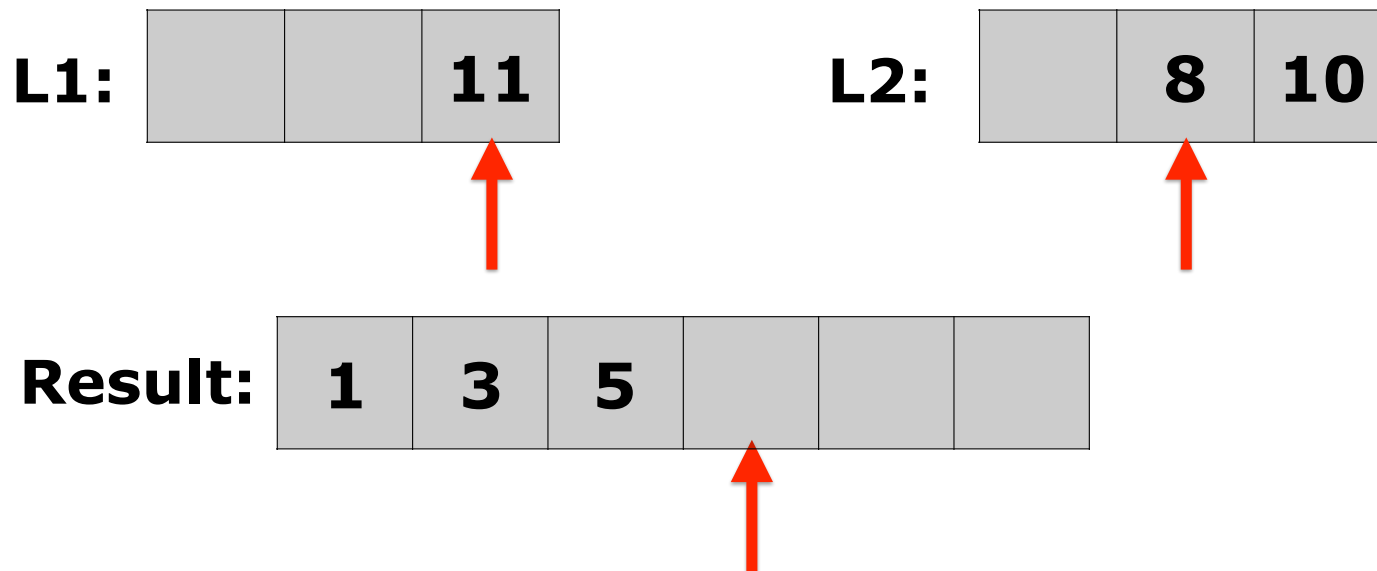
# Merge Sort

- Merging two sorted lists is easy:



# Merge Sort

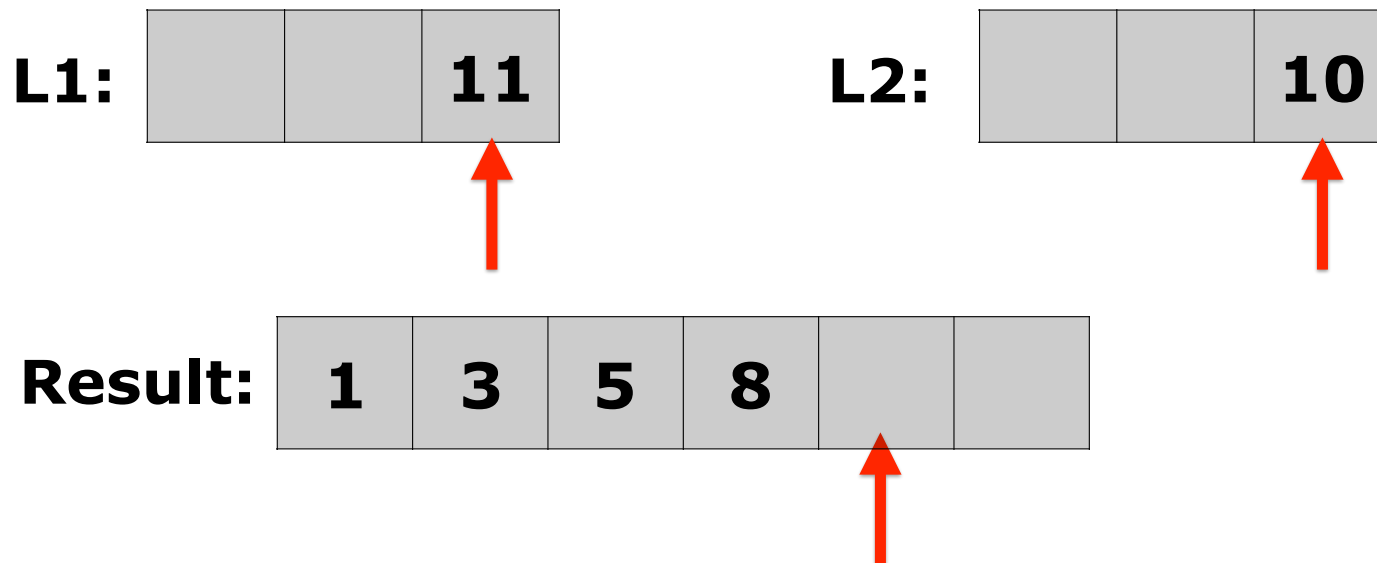
- Merging two sorted lists is easy:





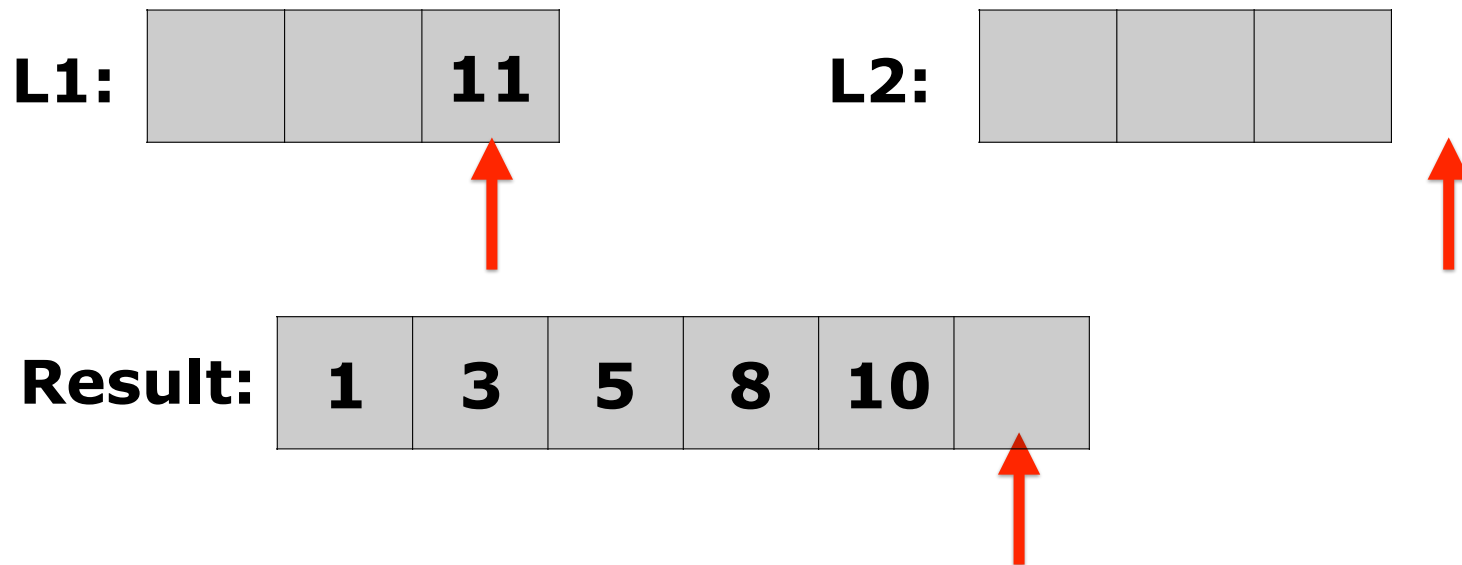
# Merge Sort

- Merging two sorted lists is easy:



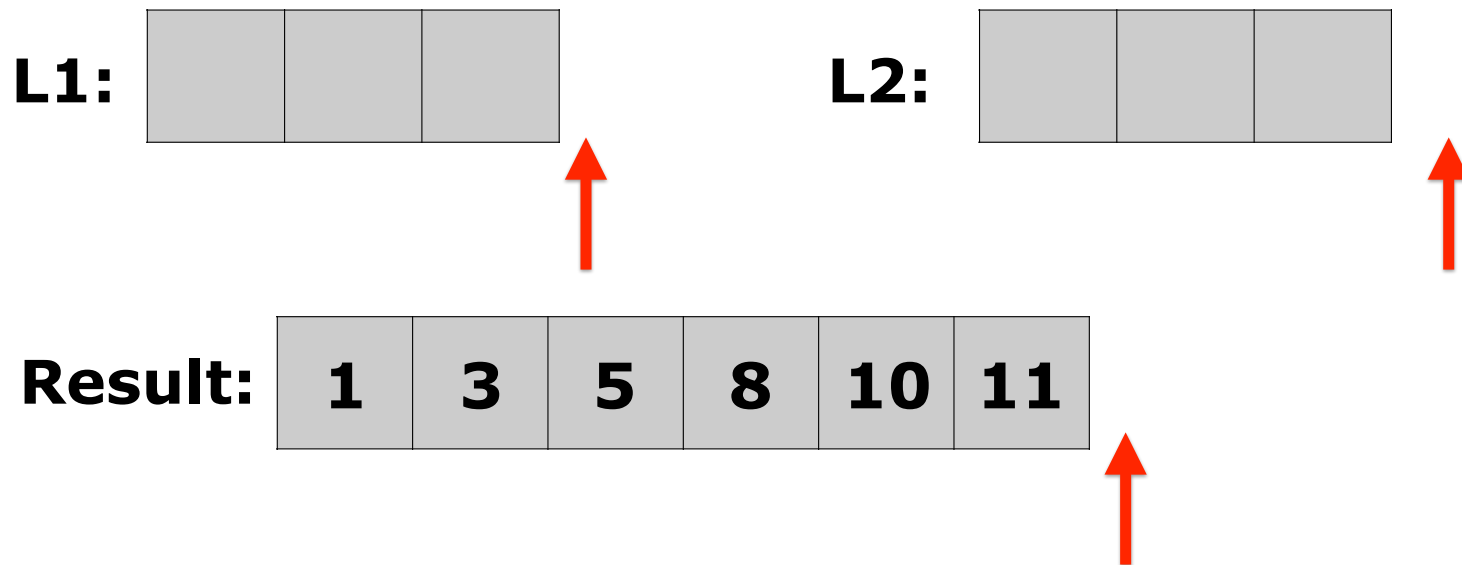
# Merge Sort

- Merging two sorted lists is easy:



# Merge Sort

- Merging two sorted lists is easy:



# Merge Sort

- Full algorithm:
  - Divide the unsorted list into  $n$  sublists, each containing 1 element (a list of 1 element is considered sorted).
  - Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.



# Merge Sort Code (Recursive!)

```
// Rearranges the elements of v into sorted order using
// the merge sort algorithm.
void mergeSort(Vector<int> &vec) {
    int n = vec.size();
    if (n <= 1) return;
    Vector<int> v1;
    Vector<int> v2;
    for (int i=0; i < n; i++) {
        if (i < n / 2) {
            v1.add(vec[i]);
        } else {
            v2.add(vec[i]);
        }
    }
    mergeSort(v1);
    mergeSort(v2);
    vec.clear();
    merge(vec, v1, v2);
}
```



# Merge Halves Code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted, and vec is empty
void merge(Vector<int> &vec, Vector<int> &v1, Vector<int> &v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] < v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    while (p1 < n1) {
        vec.add(v1[p1++]);
    }
    while (p2 < n2) {
        vec.add(v2[p2++]);
    }
}
```



# Merge Sort: Full Example

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>	<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------



# Merge Sort: Full Example

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>	<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>
-----------	----------	-----------	-----------

<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	-----------	-----------	----------	----------





# Merge Sort: Full Example

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>	<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>
-----------	----------	-----------	-----------

<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>
-----------	----------

<b>86</b>	<b>15</b>
-----------	-----------

<b>58</b>	<b>35</b>
-----------	-----------

<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	----------



# Merge Sort: Full Example

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>	<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>
-----------	----------	-----------	-----------

<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>
-----------	----------

<b>86</b>	<b>15</b>
-----------	-----------

<b>58</b>	<b>35</b>
-----------	-----------

<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	----------

<b>99</b>	<b>6</b>
-----------	----------

<b>86</b>	<b>15</b>
-----------	-----------

<b>58</b>	<b>35</b>
-----------	-----------

<b>86</b>
-----------

<b>4</b>	<b>0</b>
----------	----------



# Merge Sort: Full Example

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>	<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>
-----------	----------	-----------	-----------

<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	-----------	-----------	----------	----------

<b>99</b>	<b>6</b>
-----------	----------

<b>86</b>	<b>15</b>
-----------	-----------

<b>58</b>	<b>35</b>
-----------	-----------

<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	----------

<b>99</b>	<b>6</b>
-----------	----------

<b>86</b>	<b>15</b>
-----------	-----------

<b>58</b>	<b>35</b>
-----------	-----------

<b>86</b>
-----------

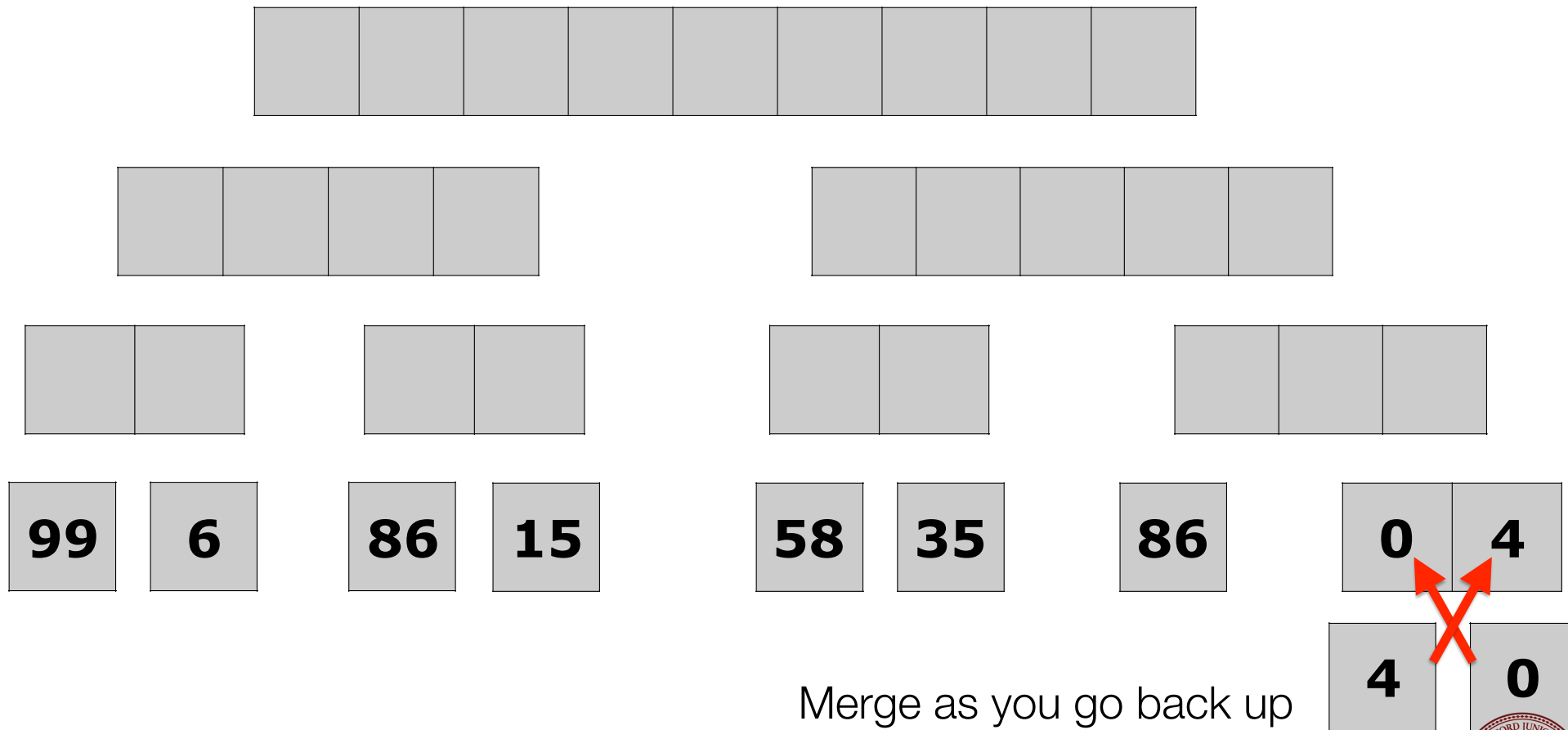
<b>4</b>	<b>0</b>
----------	----------

<b>4</b>
----------

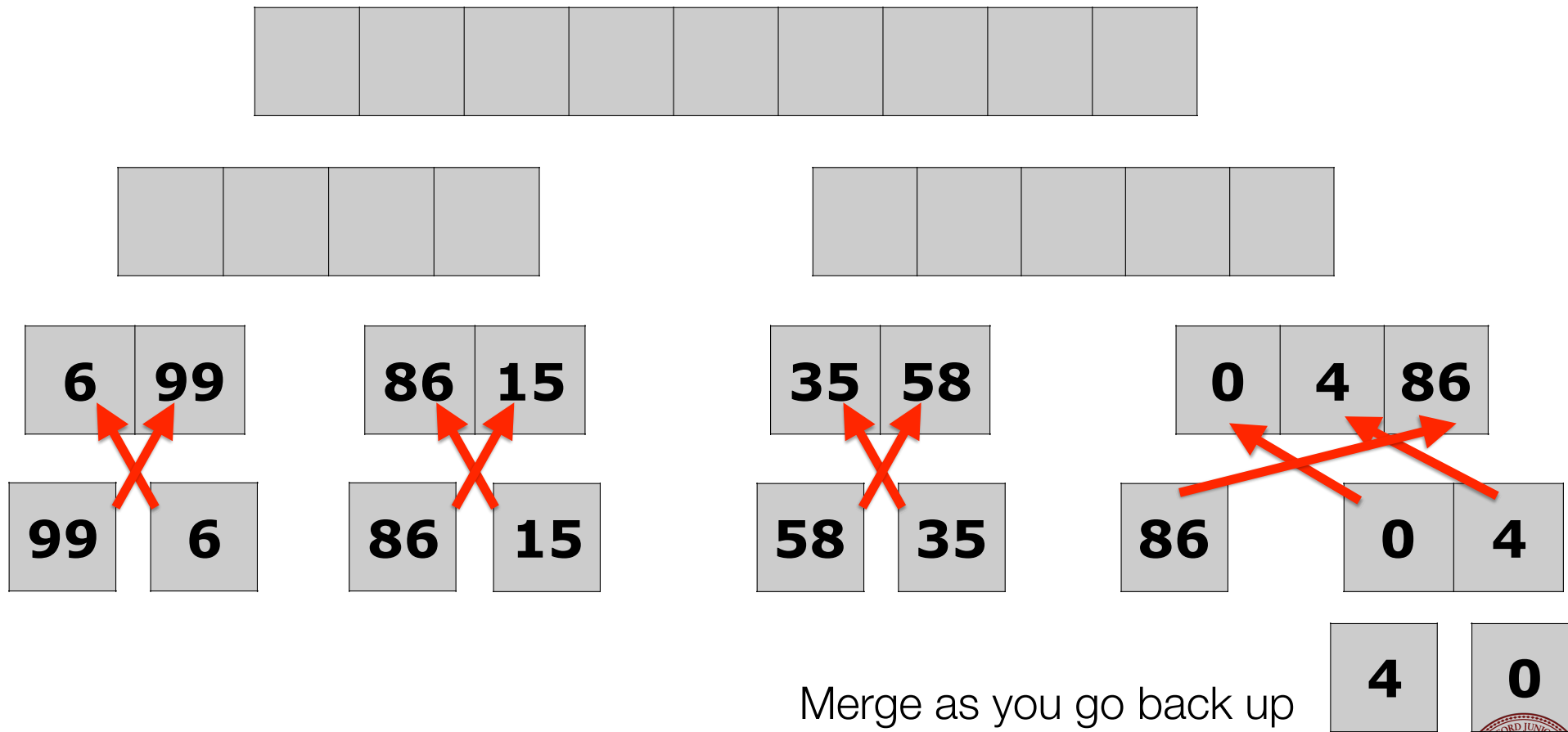
<b>0</b>
----------



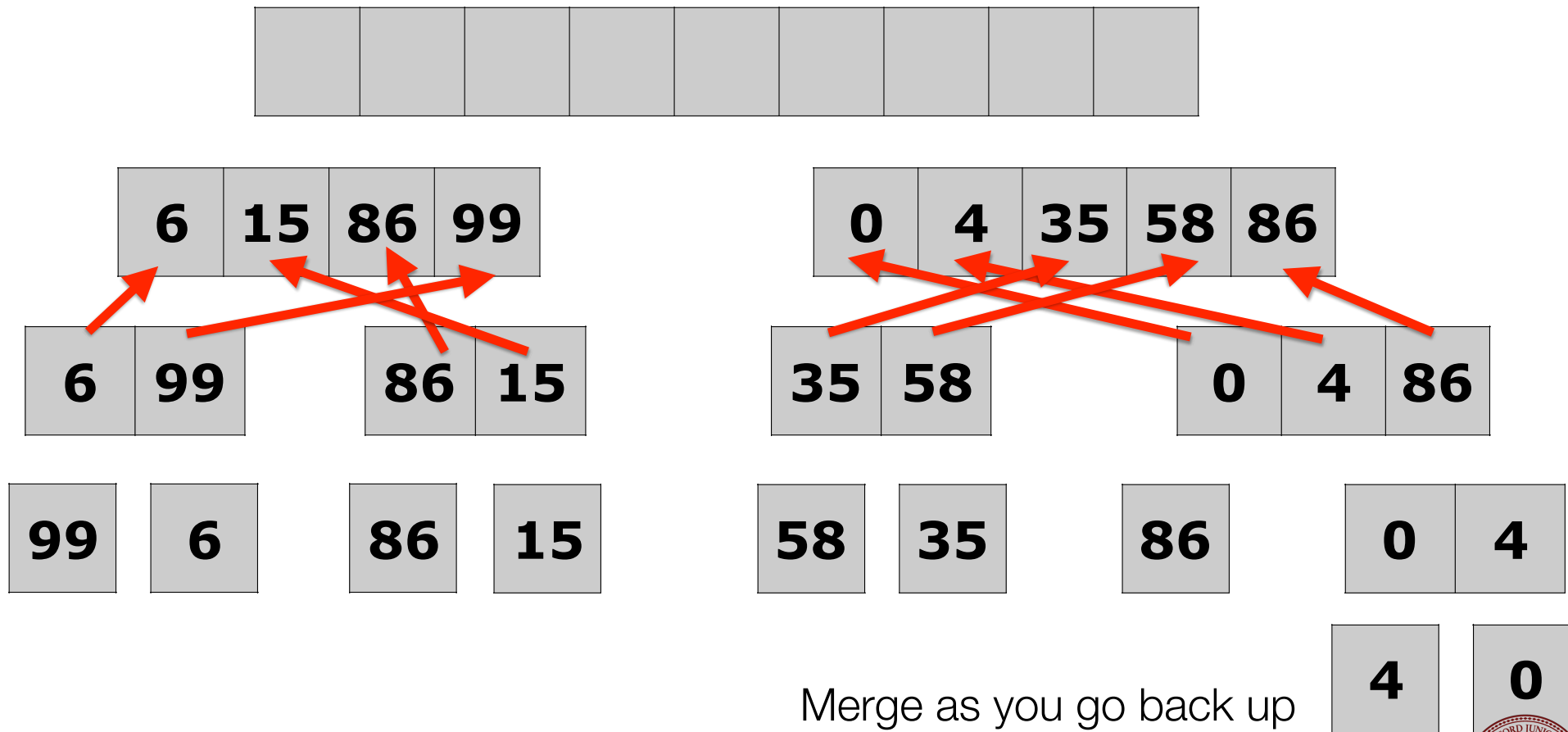
# Merge Sort: Full Example



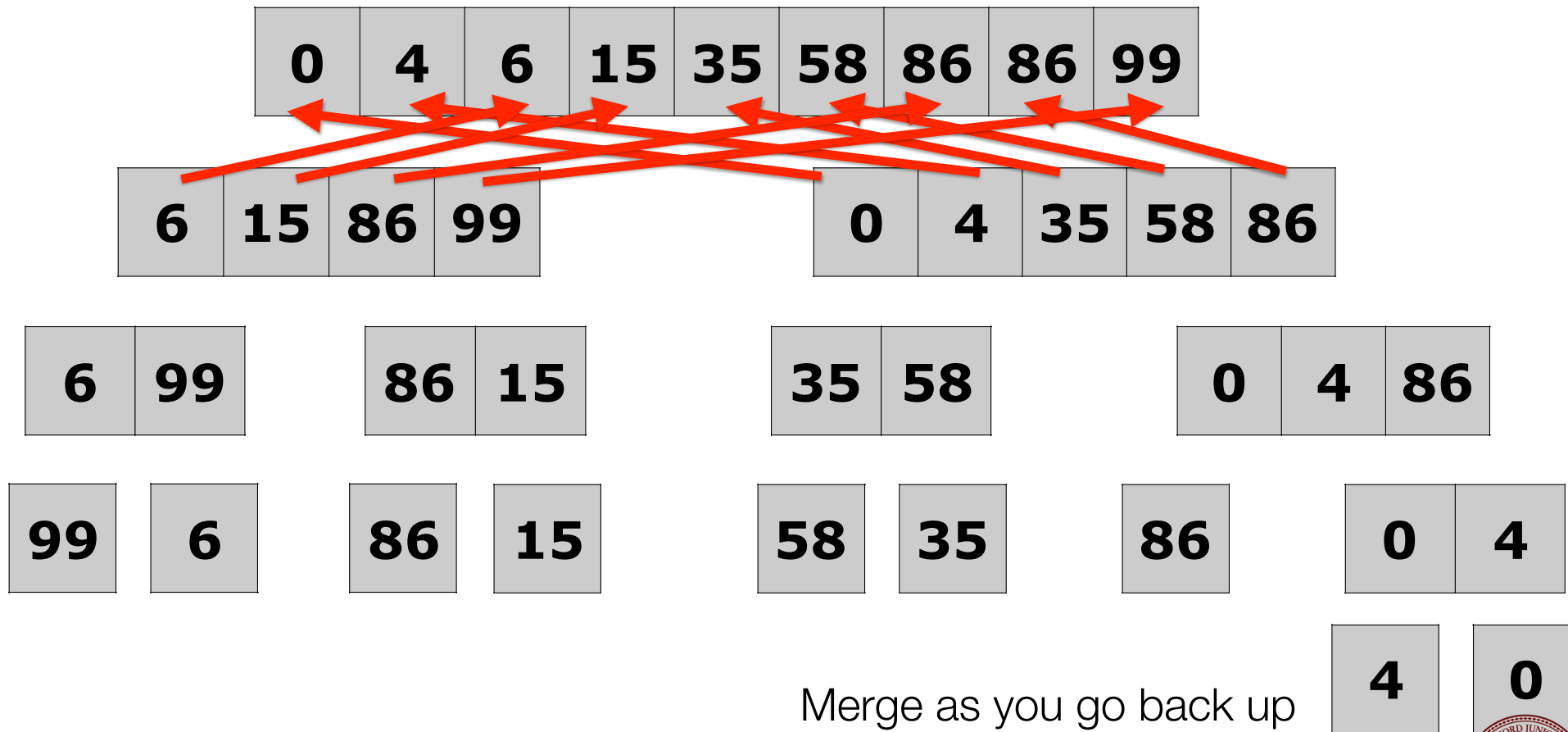
# Merge Sort: Full Example



# Merge Sort: Full Example



# Merge Sort: Full Example



# Merge Sort: Space Complexity

<b>0</b>	<b>4</b>	<b>6</b>	<b>15</b>	<b>35</b>	<b>58</b>	<b>86</b>	<b>86</b>	<b>99</b>
----------	----------	----------	-----------	-----------	-----------	-----------	-----------	-----------

- Merge Sort can be completed in place, but
  - It takes more time because elements may have to be shifted often
- It can also use “double storage” with a temporary array.
  - This is fast, because no elements need to be shifted
  - It takes double the memory, which makes it inefficient for in-memory sorts.





# Merge Sort: Time Complexity

<b>0</b>	<b>4</b>	<b>6</b>	<b>15</b>	<b>35</b>	<b>58</b>	<b>86</b>	<b>86</b>	<b>99</b>
----------	----------	----------	-----------	-----------	-----------	-----------	-----------	-----------

- The Double Memory merge sort has a worst-case time complexity of  **$O(n \log n)$**  (this is great!)
- Best case is also  **$O(n \log n)$**
- Average case is  **$O(n \log n)$**
  
- *Note:* We would like you to understand this analysis (and know the outcomes above), but it is not something we will expect you to reinvent on the midterm.



# Is our Merge Sort "stable"?

- A "stable sort" keeps the original order of the same values in the same order. E.g.,

<b>99</b>	<b>6</b>	<b>86</b>	<b>15</b>	<b>58</b>	<b>35</b>	<b>86</b>	<b>4</b>	<b>0</b>
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

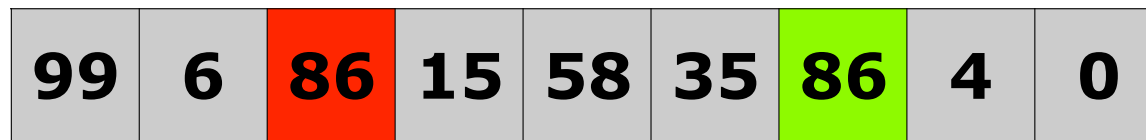
this 86

will end up  
ahead of this 86



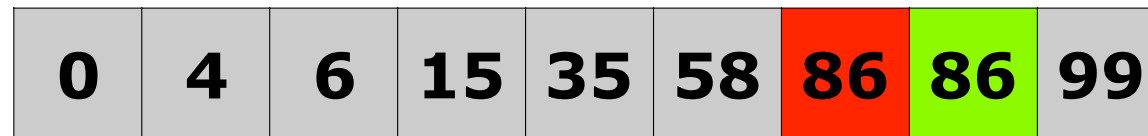
# Is our Merge Sort "stable"?

- A "stable sort" keeps the original order of the same values in the same order. E.g.,



this 86

will end up  
ahead of this 86



Who cares? It's just a number!



# Is our Merge Sort "stable"?

What if we were sorting linked vectors? What if we first sorted by alpha below, then by num...

num	99	6	86	15	58	35	86	4	0
alpha	A	B	C	D	E	F	G	H	I

We might care! If we are sorting first names with last names, maybe we want all the last names to be in order.

num	0	4	6	15	35	58	86	86	99
alpha	I	H	B	D	F	E	C	G	A

num	0	4	6	15	35	58	86	86	99
alpha	I	H	B	D	F	E	G	C	A



# Is our Merge Sort "stable"?

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted, and vec is empty
void merge(Vector<int> &vec, Vector<int> &v1, Vector<int> &v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] < v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    while (p1 < n1) {
        vec.add(v1[p1++]);
    }
    while (p2 < n2) {
        vec.add(v2[p2++]);
    }
}
```



# Is our Merge Sort "stable"?

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted, and vec is empty
void merge(Vector<int> &vec, Vector<int> &v1, Vector<int> &v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] < v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    while (p1 < n1) {
        vec.add(v1[p1++]);
    }
    while (p2 < n2) {
        vec.add(v2[p2++]);
    }
}
```

**Nope, not stable!**



# Is our Merge Sort "stable"?

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted, and vec is empty
void merge(Vector<int> &vec, Vector<int> &v1, Vector<int> &v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] <= v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    while (p1 < n1) {
        vec.add(v1[p1++]);
    }
    while (p2 < n2) {
        vec.add(v2[p2++]);
    }
}
```

**But we can make it stable**



# Sorts

Insertion Sort  
Selection Sort  
Merge Sort  
Quicksort





# Quicksort

- Quicksort is a sorting algorithm that is often faster than most other types of sorts.
- However, although it has an average  **$O(n \log n)$**  time complexity, it also has a worst-case  **$O(n^2)$**  time complexity, though this rarely occurs.



# Quicksort

- Quicksort is another divide-and-conquer algorithm.
- The basic idea is to **divide** a list into two smaller sub-lists: **the low elements and the high elements**. Then, the algorithm can recursively sort the sub-lists.



# Quicksort Algorithm

- **Pick an element**, called a **pivot**, from the list
- **Reorder** the list so that all elements with **values less than the pivot come before the pivot**, while all elements with values **greater than the pivot come after it**. After this partitioning, the pivot is in its final position. This is called the partition operation.
- **Recursively apply the above steps to the sub-list of elements** with smaller values and separately to the sub-list of elements with greater values.
- The **base case** of the recursion is for **lists of 0 or 1** elements, which do not need to be sorted.



# Quicksort Algorithm

- We have two ways to perform quicksort:
  - The **naive** algorithm: create new lists for each sub-sort, leading to an overhead of  $n$  additional memory.
  - The **in-place** algorithm, which swaps elements.



# Quicksort Algorithm: Naive

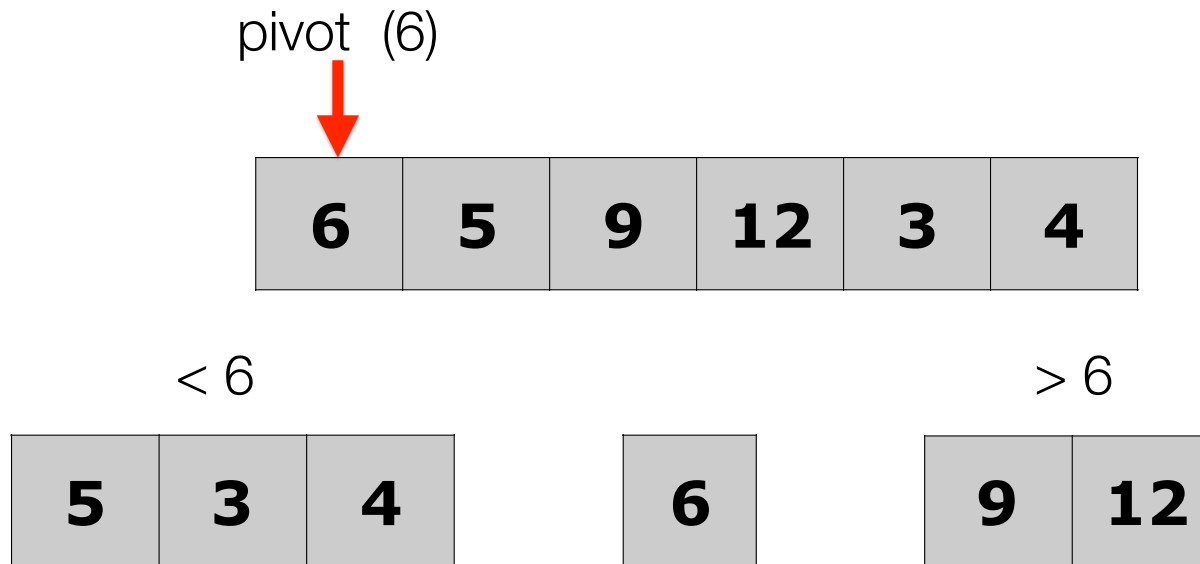
pivot (6)



<b>6</b>	<b>5</b>	<b>9</b>	<b>12</b>	<b>3</b>	<b>4</b>
----------	----------	----------	-----------	----------	----------



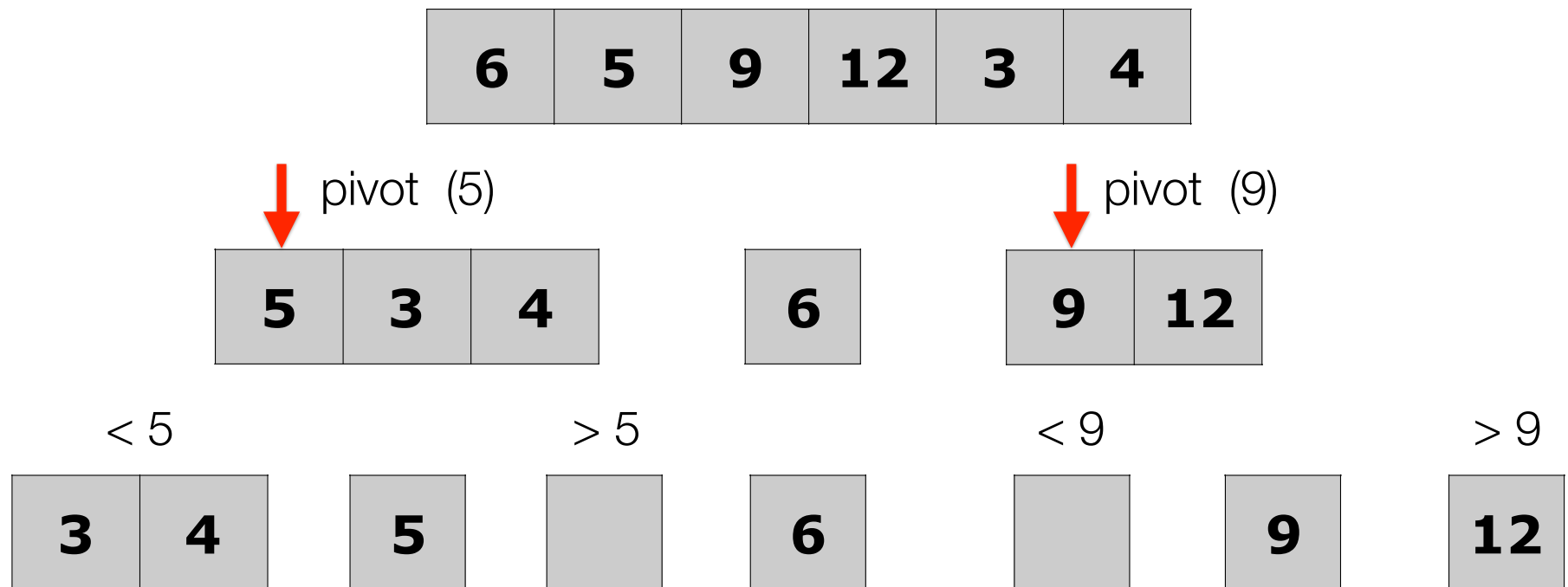
# Quicksort Algorithm: Naive



Partition into two new lists -- less than the pivot on the left, and greater than the pivot on the right. Even if all elements go into one list, that was just a poor partition.



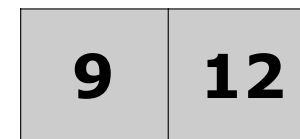
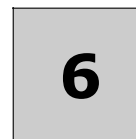
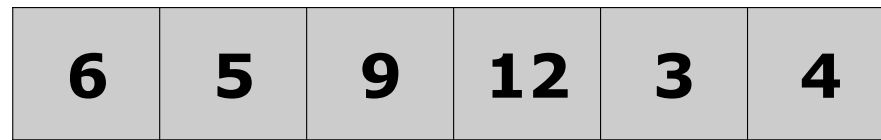
# Quicksort Algorithm: Naive



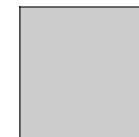
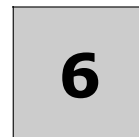
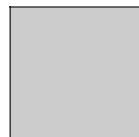
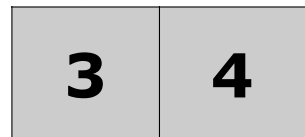
Keep partitioning the sub-lists



# Quicksort Algorithm: Naive

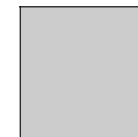
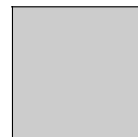


↓ pivot (3)



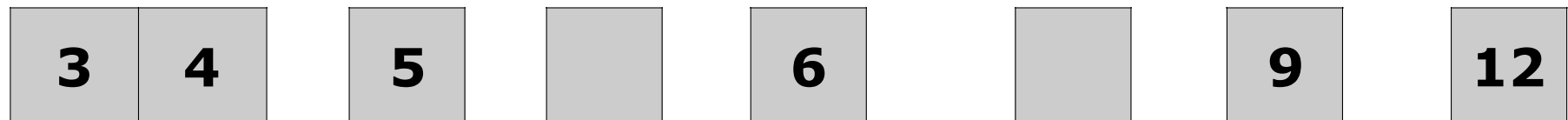
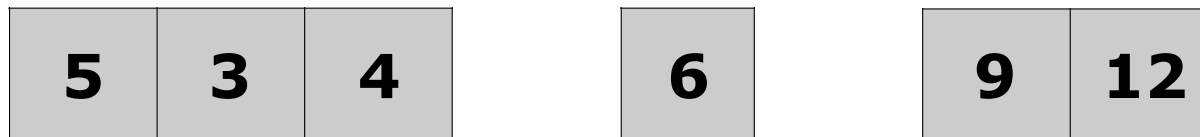
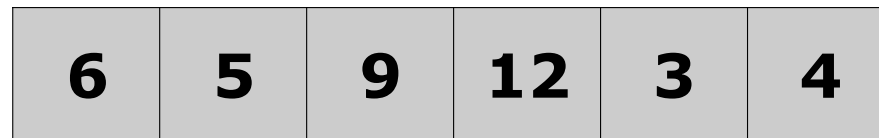
< 3

> 3





# Quicksort Algorithm: Naive



# Quicksort Algorithm: Naive Code

```
Vector<int> naiveQuickSortHelper(Vector<int> v) { // not passed by reference!  
    // base case: list of 0 or 1  
    if (v.size() < 2) {  
        return v;  
    }  
    int pivot = v[0];    // choose pivot to be left-most element  
  
    // create two new vectors to partition into  
    Vector<int> left, right;  
  
    // put all elements <= pivot into left, and all elements > pivot into right  
    for (int i=1; i<v.size(); i++) {  
        if (v[i] <= pivot) {  
            left.add(v[i]);  
        }  
        else {  
            right.add(v[i]);  
        }  
    }  
    left = naiveQuickSortHelper(left); // recursively handle the left  
    right = naiveQuickSortHelper(right); // recursively handle the right  
  
    left.add(pivot); // put the pivot at the end of the left  
    return left + right; // return the combination of left and right  
}
```



# Quicksort Algorithm: In-Place

0	1	2	3	4	5	6	7
56	25	37	58	95	19	73	30

In-place, recursive algorithm:

```
int quickSort(Vector<int> &v, int start, int finish);
```


- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the pivot with the element where the left/right cross, unless it happens to be the pivot.

**This is best described with a detailed example...**



# Quicksort Algorithm: In-Place

0	1	2	3	4	5	6	7
56	25	37	58	95	19	73	30

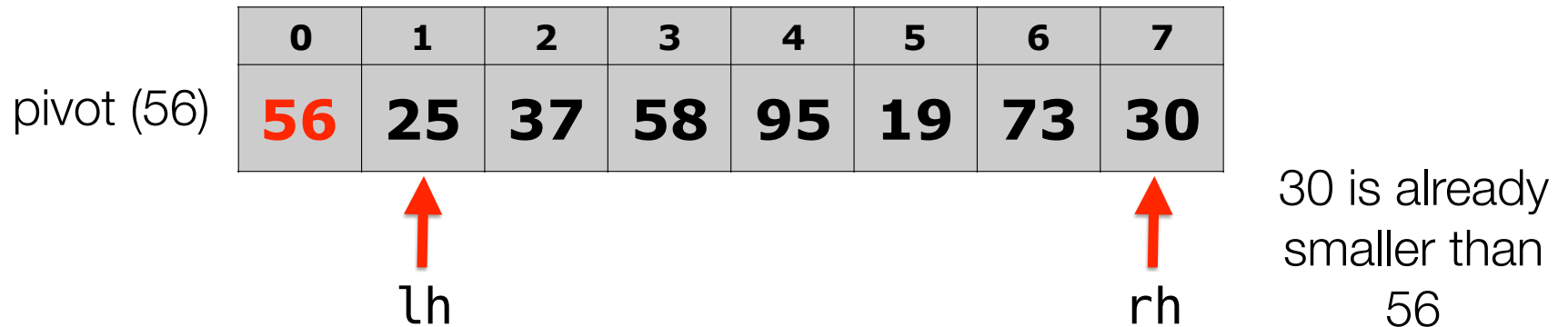
  
pivot (56)

```
quickSort(vec, 0, 7);
```

- **Pick your pivot as the left element (might not be a good choice...)**
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.



# Quicksort Algorithm: In-Place



```
quickSort(vec, 0, 7);
```

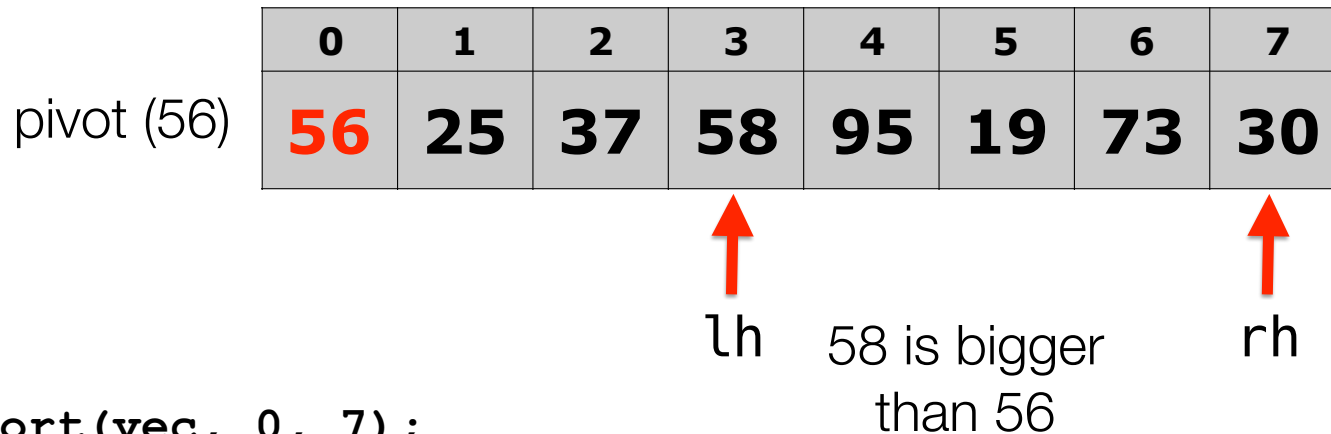
- Pick your pivot as the left element (might not be a good choice...)
- **Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.**
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.







# Quicksort Algorithm: In-Place



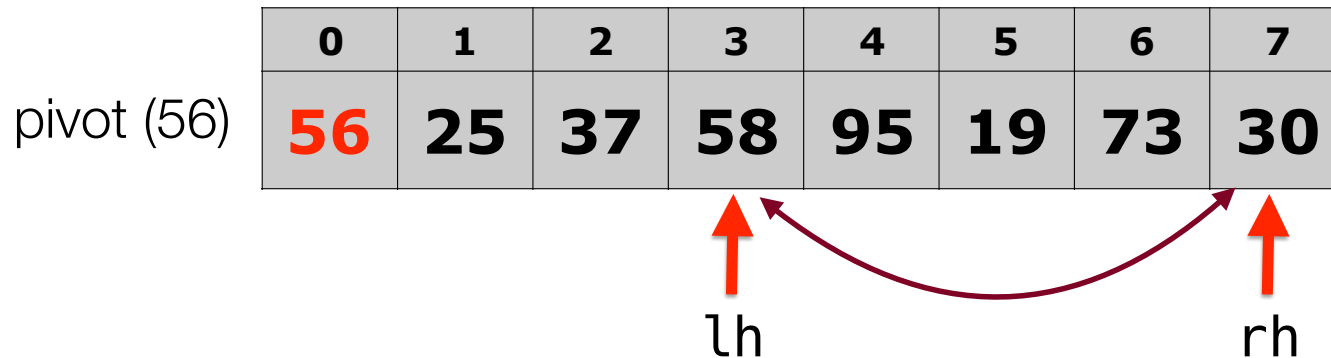
```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- **Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.**
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.





# Quicksort Algorithm: In-Place

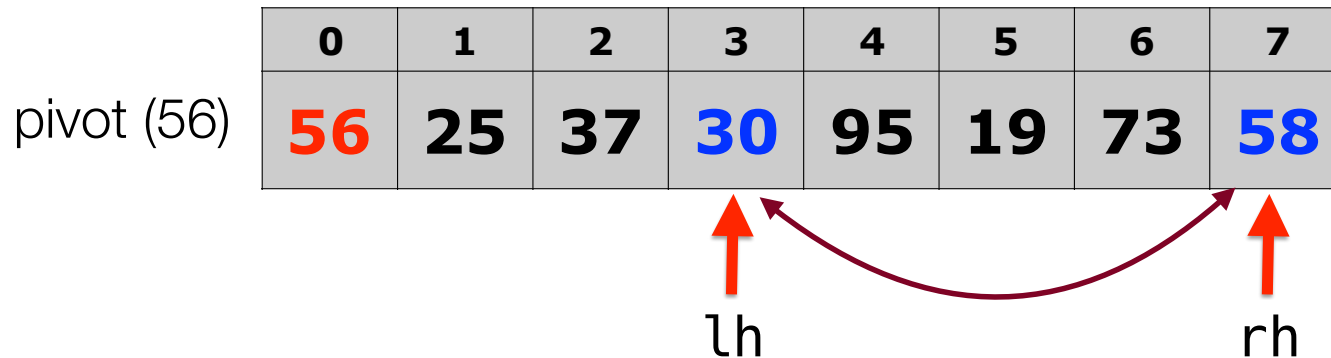


```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- **Swap the two elements where the left/right cross, unless the pivot is the smallest.**
- Repeat the traversals until they cross, at which point you swap that element with the pivot



# Quicksort Algorithm: In-Place



```
quickSort(vec, 0, 7);
```

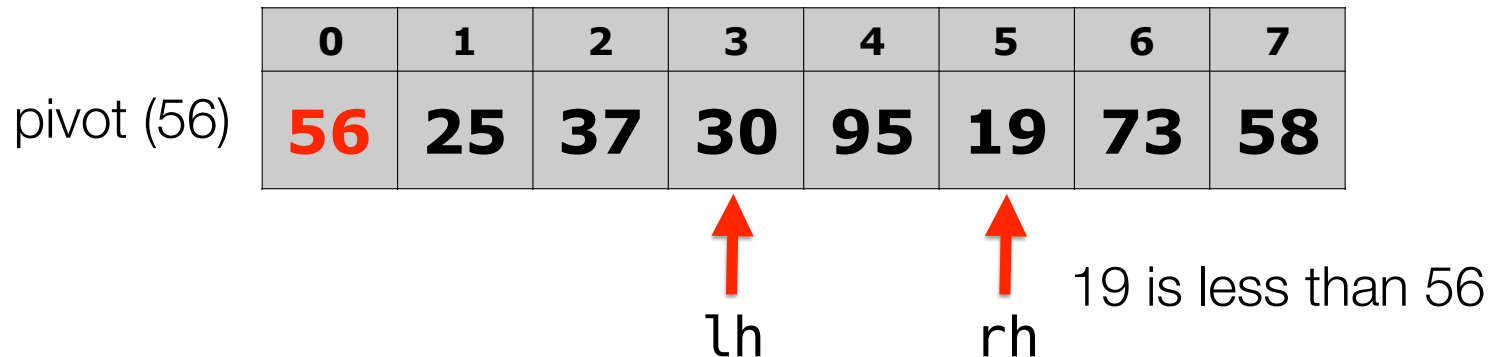
- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- **Swap the two elements where the left/right cross, unless the pivot is the smallest.**
- Repeat the traversals until they cross, at which point you swap that element with the pivot







# Quicksort Algorithm: In-Place



```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- **Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.**
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.



# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
<b>56</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>95</b>	<b>19</b>	<b>73</b>	<b>58</b>

↑                      ↑  
lh                      rh

```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- **Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.**
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.



# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
<b>56</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>95</b>	<b>19</b>	<b>73</b>	<b>58</b>

lh      rh

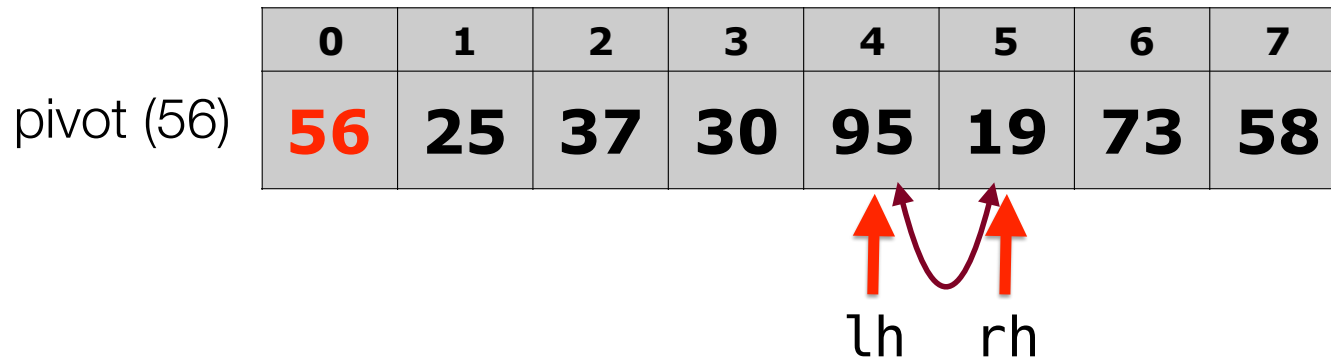
95 is greater than 56

```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- **Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.**
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.



# Quicksort Algorithm: In-Place



```
quickSort(vec, 0, 7);
```

95 is greater than 56

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- **Swap the two elements where the left/right cross, unless the pivot is the smallest.**
- Repeat the traversals until they cross, at which point you swap that element with the pivot





# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
<b>56</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>19</b>	<b>95</b>	<b>73</b>	<b>58</b>

lh rh

```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- **Swap the two elements where the left/right cross, unless the pivot is the smallest.**
- Repeat the traversals until they cross, at which point you swap that element with the pivot



# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
<b>56</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>19</b>	<b>95</b>	<b>73</b>	<b>58</b>

lh rh

```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- **Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.**
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.



# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
<b>56</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>19</b>	<b>95</b>	<b>73</b>	<b>58</b>

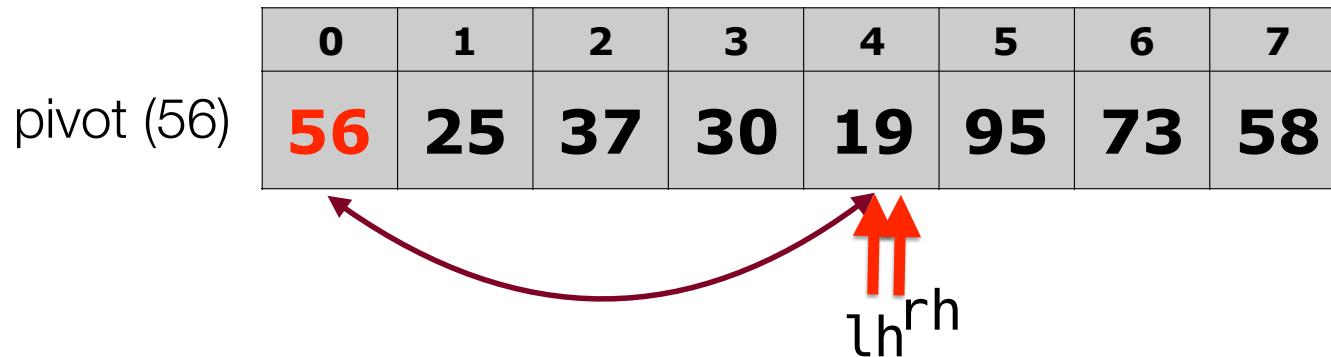
↑↑  
lh<sup>rh</sup>

```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- **Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.**
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- Repeat the traversals until they cross, at which point you swap that element with the pivot.



# Quicksort Algorithm: In-Place



```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- **Repeat the traversals until they cross, at which point you swap that element with the pivot.**



# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
19	25	37	30	56	95	73	58

↑↑  
lh<sup>rh</sup>

```
quickSort(vec, 0, 7);
```

- Pick your pivot as the left element (might not be a good choice...)
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot, or it hits the left.
- Traverse the list from the beginning (left, after pivot) forwards until the value should be to the *right* of the pivot, or until it hits the right.
- Swap the two elements where the left/right cross, unless the pivot is the smallest.
- **Repeat the traversals until they cross, at which point you swap that element with the pivot.**



# Quicksort Algorithm: In-Place

pivot (56)

0	1	2	3	4	5	6	7
<b>19</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>56</b>	<b>95</b>	<b>73</b>	<b>58</b>

↑↑  
 $l^h$   $r^h$

```
quickSort(vec, 0, 7);
```

- The partitioning step has completed! The elements to the left of 56 are smaller, and the elements to the right are bigger!
- The partitioning step returns the "boundary" value (index 4, in this case), and we can now sort each sub-part of the vector:

```
quickSort(vec, 0, 3);
```

```
quickSort(vec, 4, 7);
```

If start is ever bigger than finish, we just return!



# Quicksort Algorithm: Big-O

0	1	2	3	4	5	6	7
<b>19</b>	<b>25</b>	<b>37</b>	<b>30</b>	<b>56</b>	<b>95</b>	<b>73</b>	<b>58</b>

- Best-case time complexity:  $O(n \log n)$
- Worst-case time complexity:  $O(n^2)$
- Average time complexity:  $O(n \log n)$
- Space complexity: naive:  $O(n)$  extra, in-place:  $O(\log n)$  extra (because of recursion)
- Stable?



# Quicksort In-place Code

```
/*
 * Rearranges the elements of v into sorted order using
 * a recursive quick sort algorithm.
 */
void quicksort(Vector<int> &vec) {
    quicksort(vec, 0, vec.size() - 1);
}

void quicksort(Vector<int> &vec, int start, int finish) {
    if (start >= finish) return;
    int boundary = partition(vec, start, finish);
    quicksort(vec, start, boundary - 1);
    quicksort(vec, boundary + 1, finish);
}
```

We need a helper function to pass along left and right.





# Quicksort In-place Code: Partition

```
int partition(Vector<int> &vec, int start, int finish) {
    int pivot = vec[start];
    int lh = start + 1;
    int rh = finish;

    while (true) {
        while (lh < rh && vec[rh] >= pivot) rh--;
        while (lh < rh && vec[lh] < pivot) lh++;
        if (lh == rh) break;

        // swap
        int tmp = vec[lh];
        vec[lh] = vec[rh];
        vec[rh] = tmp;
    }

    if (vec[lh] >= pivot) return start;
    vec[start] = vec[lh];
    vec[lh] = pivot;
    return lh;
}
```



# Recap

Sorting Big-O Cheat Sheet			
Sort	Worst Case	Best Case	Average Case
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$



# References and Advanced Reading

## • **References:**

- [http://en.wikipedia.org/wiki/Sorting\\_algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) (excellent)
- <http://www.sorting-algorithms.com> (fantastic visualization)
- More online visualizations: <http://www.cs.usfca.edu/~galles/visualization/Algorithms.html> (excellent)
- Excellent mergesort video: <https://www.youtube.com/watch?v=GCae1WNvnZM>
- Excellent quicksort video: [https://www.youtube.com/watch?v=XE4VP\\_8Y0BU](https://www.youtube.com/watch?v=XE4VP_8Y0BU)
- Full quicksort trace: <http://goo.gl/vOgaT5>

## • **Advanced Reading:**

- YouTube video, 15 sorts in 6 minutes: <https://www.youtube.com/watch?v=kPRA0W1kECg> (fun, with sound!)
- Amazing folk dance sorts: <https://www.youtube.com/channel/UCIqiLefbVHsOAXDaxQJH7Xw>
- Radix Sort: [https://en.wikipedia.org/wiki/Radix\\_sort](https://en.wikipedia.org/wiki/Radix_sort)
- Good radix animation: <https://www.cs.auckland.ac.nz/software/AlgAnim/radixsort.html>
- Shell Sort: <https://en.wikipedia.org/wiki/Shellsort>
- Bogosort: <https://en.wikipedia.org/wiki/Bogosort>



# Extra Slides



## Why is the following nested loop $O(n^2)$ ?

```
for (int i=0; i < n; i++) {  
    for (int j=i; j < n; j++) {  
        // do stuff...  
    }  
}
```

The first time through the outer loop, there are  $n$  steps.

The second time through the outer loop, there are  $n-1$  steps.

The third time through the outer loop, there are  $n-2$  steps.

...

The last time through the outer loop, there is 1 step.



## Why is the following nested loop $O(n^2)$ ?

```
for (int i=0; i < n; i++) {  
    for (int j=i; j < n; j++) {  
        // do stuff...  
    }  
}
```

In other words, the number of total steps is:

$$n + (n-1) + (n-2) + \dots + 2 + 1 = (n + 1) * n/2 = \mathbf{n^2/2 + n/2}$$

which, by our normal rules of simplifying Big O:

$$\mathbf{n^2/2 + n/2 = O(n^2/2) = O(n^2)}$$

