

CS 106X

Lecture 11: Sorting

Friday, February 3, 2017

Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Section 10.2



Today's Topics

- Logistics
 - Tiny Feedback: *I had the thought of maybe releasing section questions ahead of sections so we could try working problems and have a chance to ask about ones we couldn't figure out or wanted to see worked in section, since there's never enough time to work all the questions in section. **Good idea!***
- 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

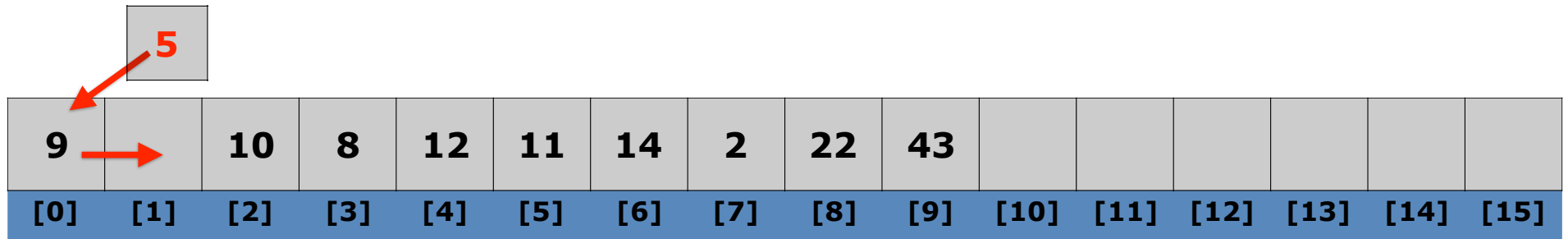
9	5	10	8	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

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



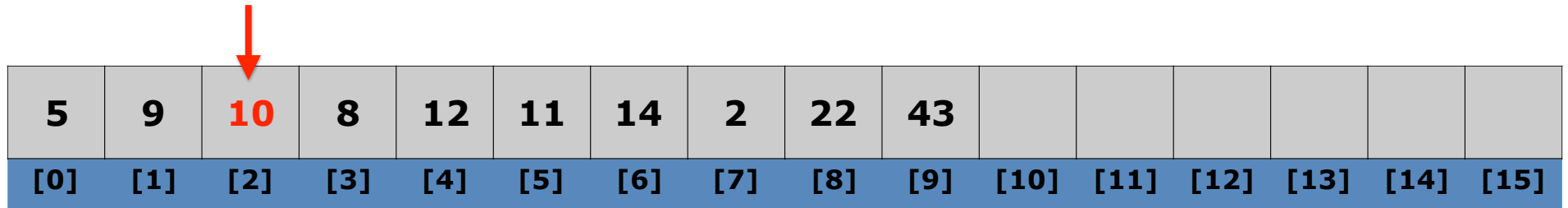
Algorithm:

- 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)



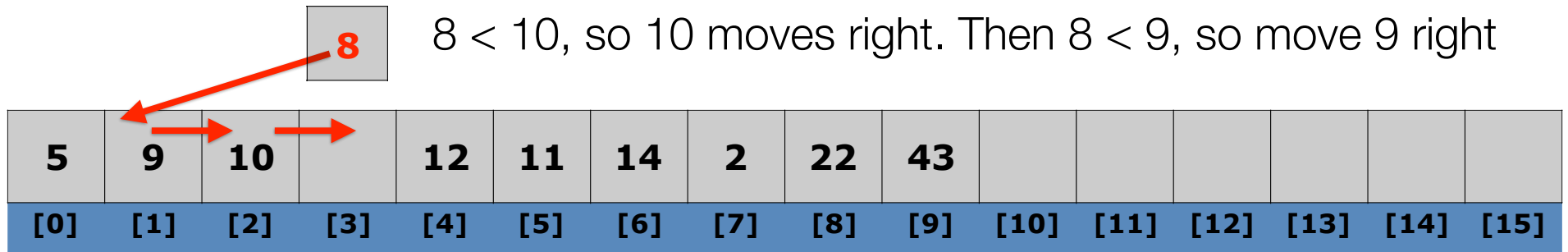
5	9	10	8	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

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)

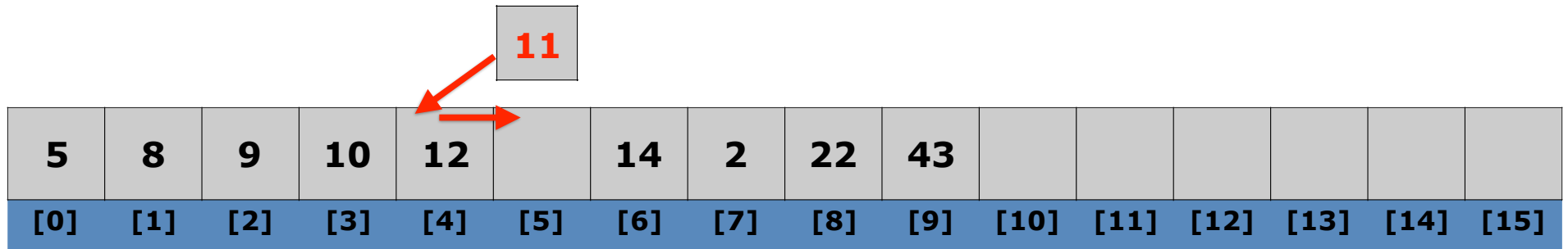
5	8	9	10	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

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)

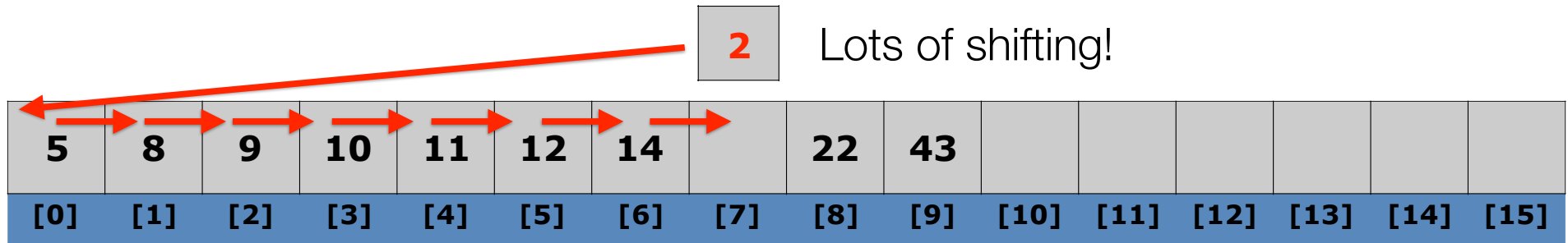
5	8	9	10	11	12	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

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?)

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



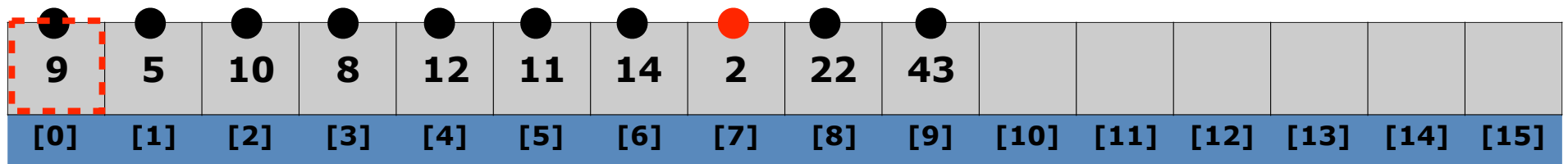
Selection Sort

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>



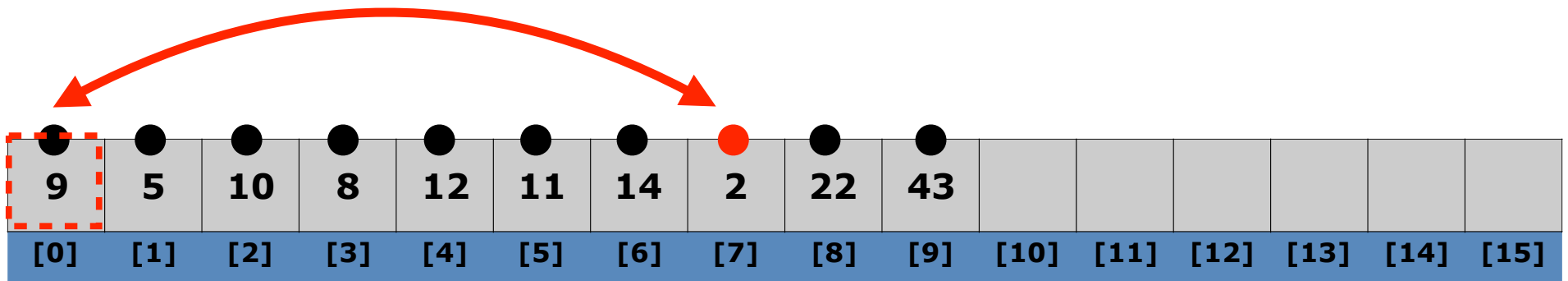
Selection Sort



- 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

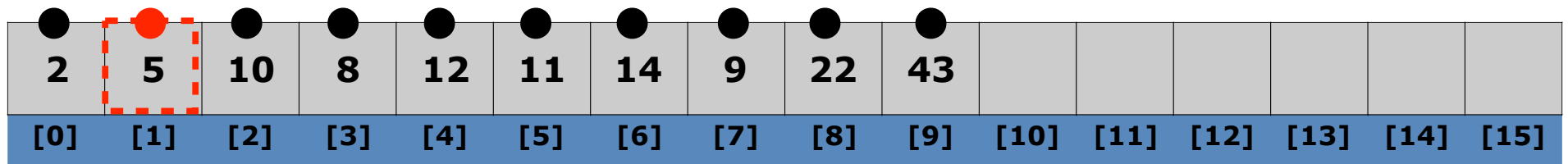


- Algorithm
 - Find the **smallest item in the list**, and exchange it with the left-most unsorted element.
 - Repeat the process from the first unsorted element.
- Selection sort is particularly slow, because it needs to go through **the entire list** each time to find the smallest item.



Selection Sort

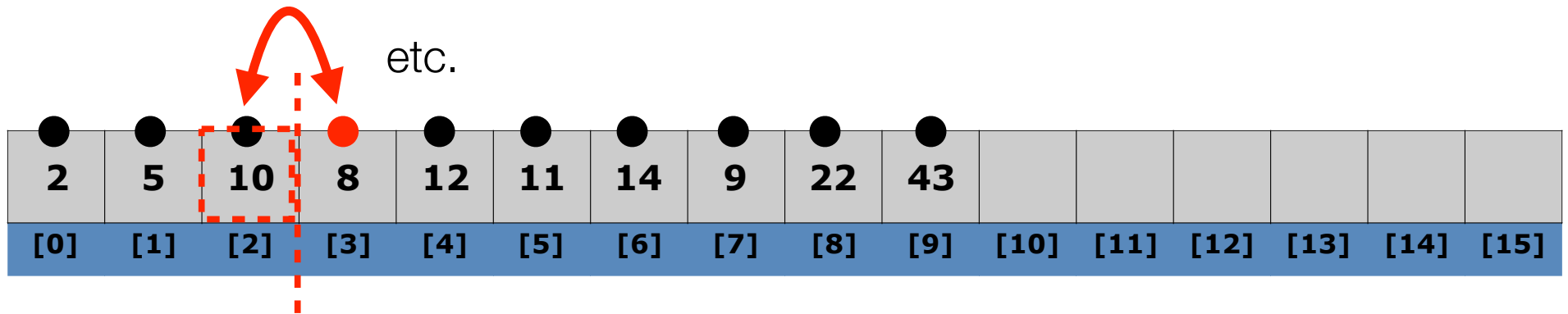
(no swap necessary)



- 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



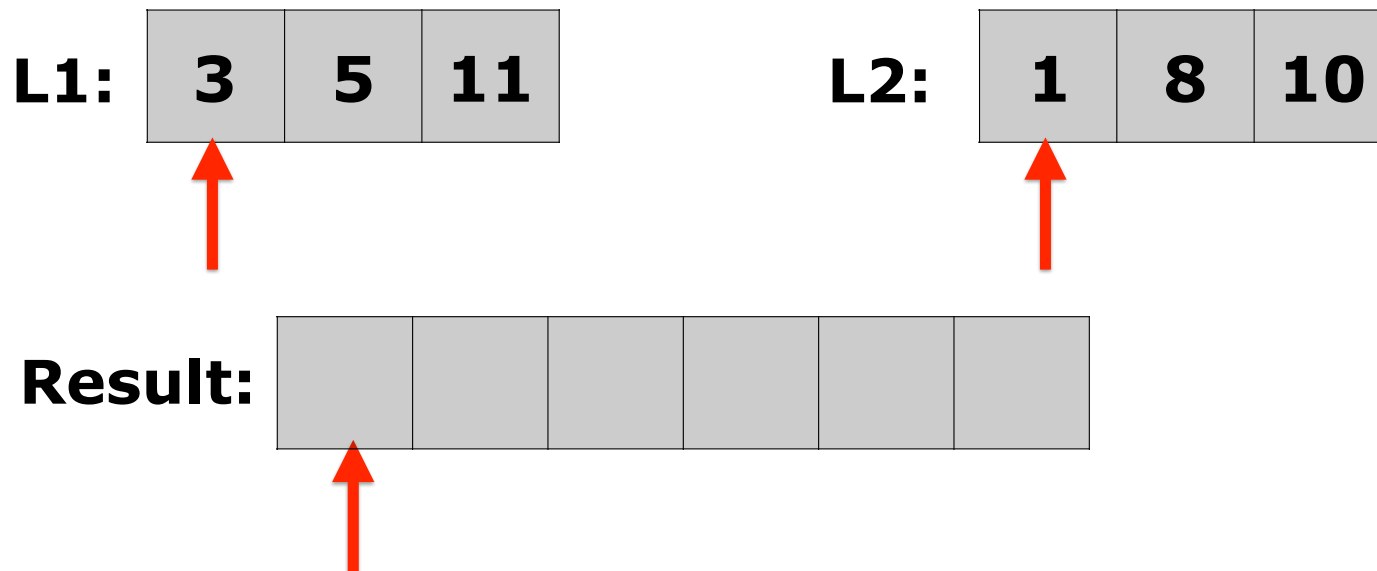
Merge Sort

- 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.,
- $L1 = \{3,5,11\}$ $L2 = \{1,8,10\}$
- $\text{merge}(L1,L2)=\{1,3,5,8,10,11\}$



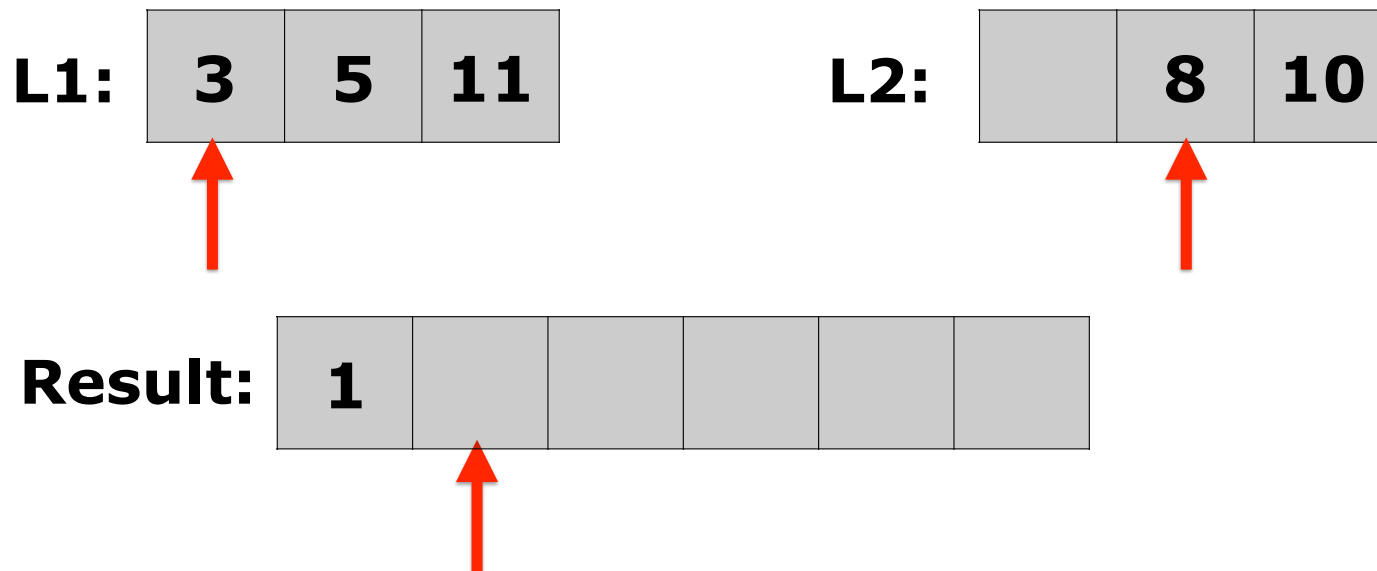
Merge Sort

- Merging two sorted lists is easy:



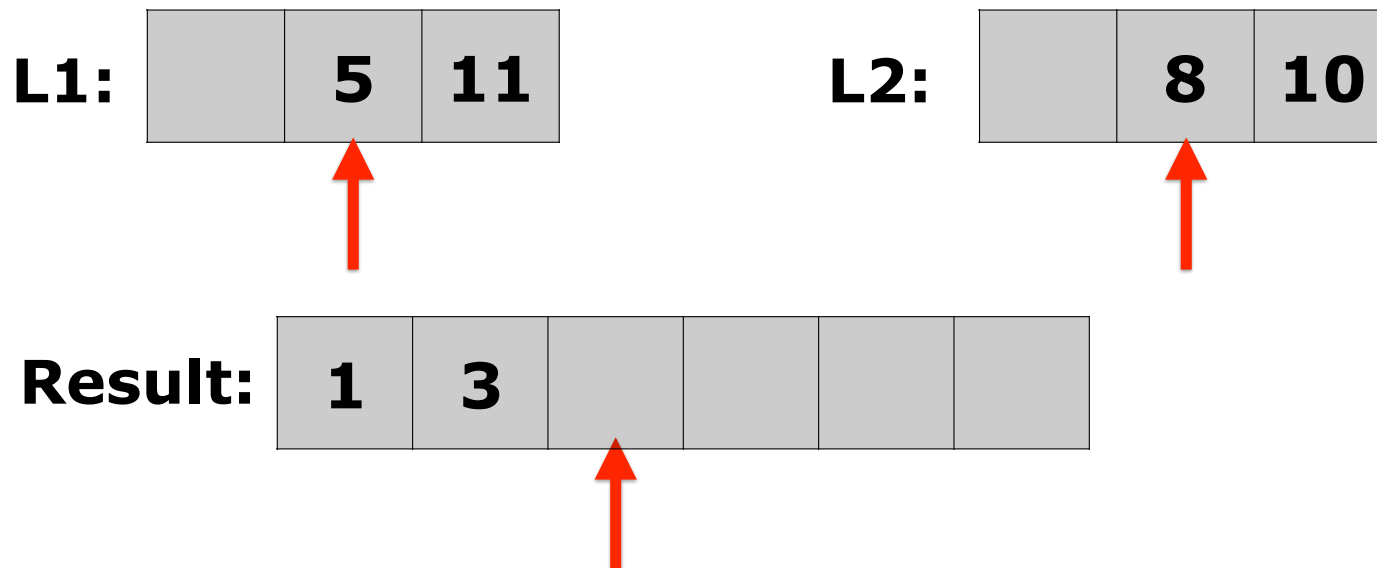
Merge Sort

- Merging two sorted lists is easy:



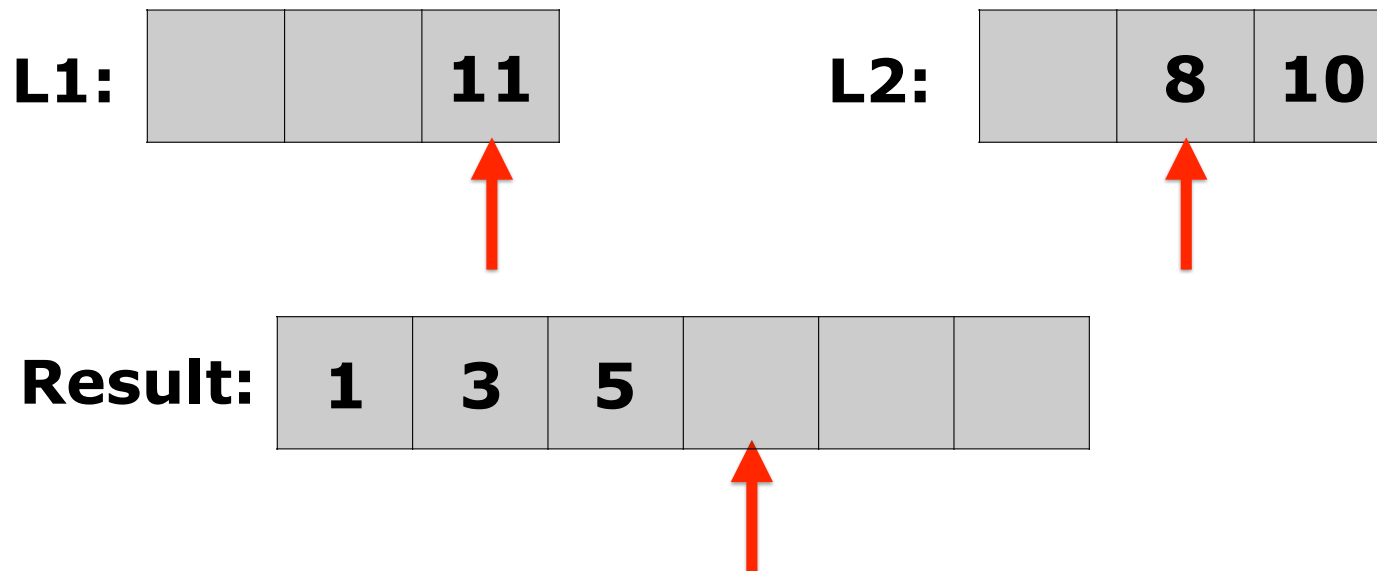
Merge Sort

- Merging two sorted lists is easy:



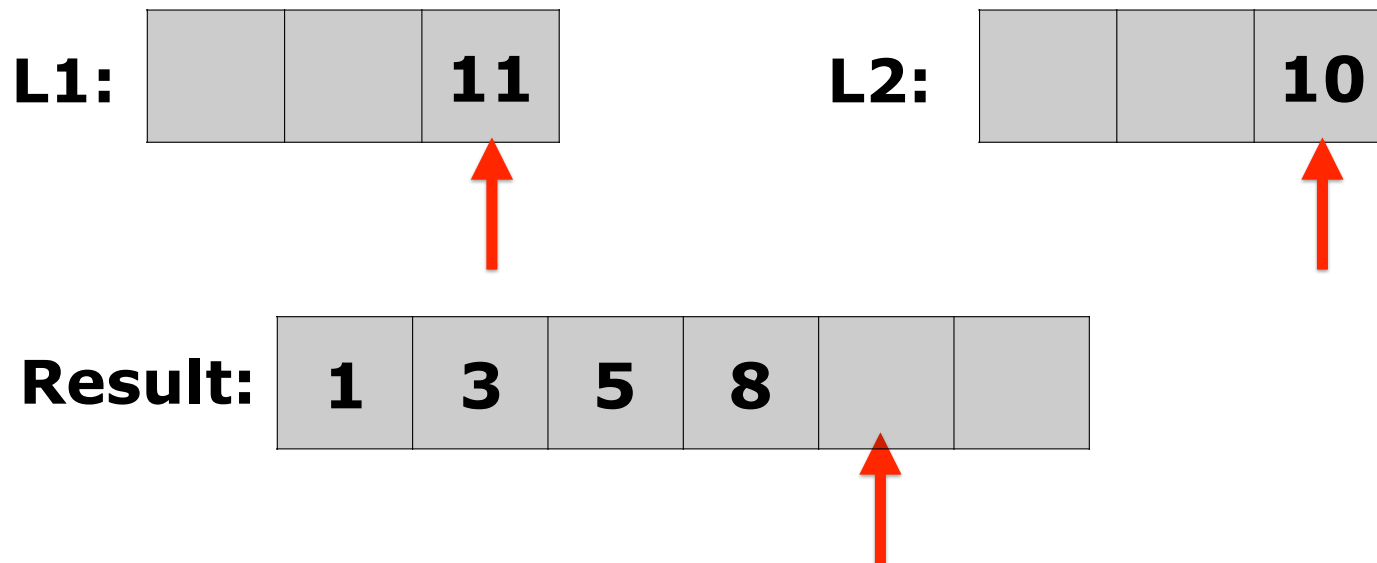
Merge Sort

- Merging two sorted lists is easy:



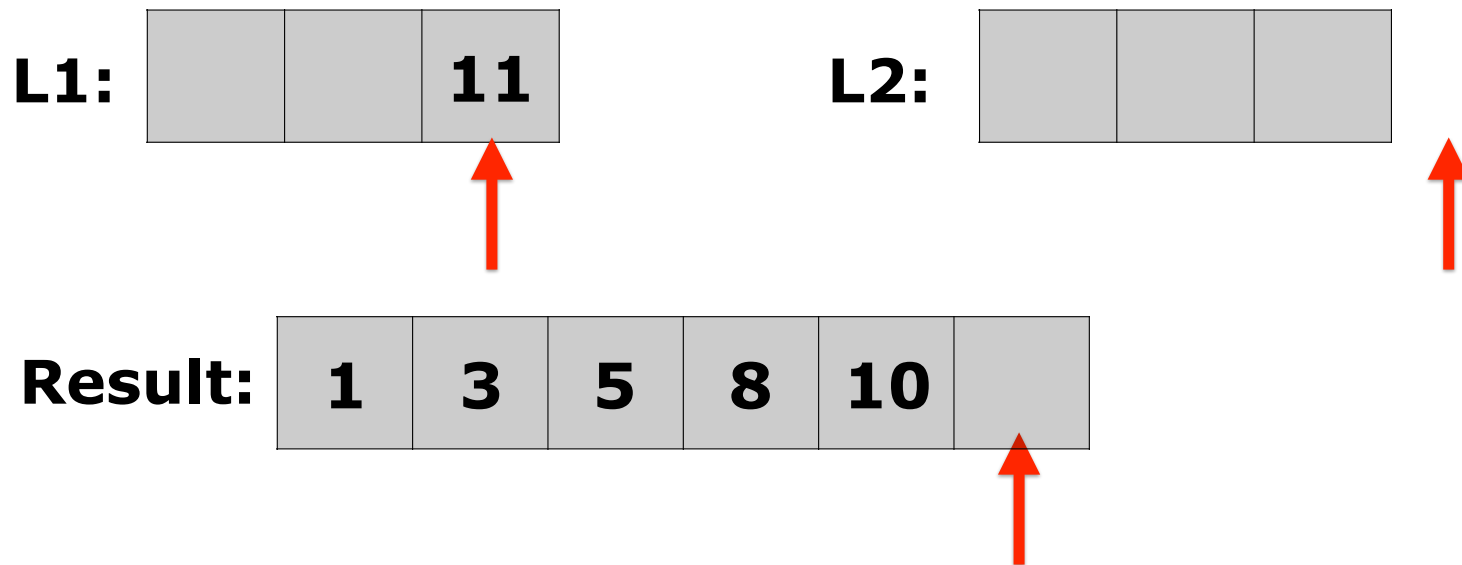
Merge Sort

- Merging two sorted lists is easy:



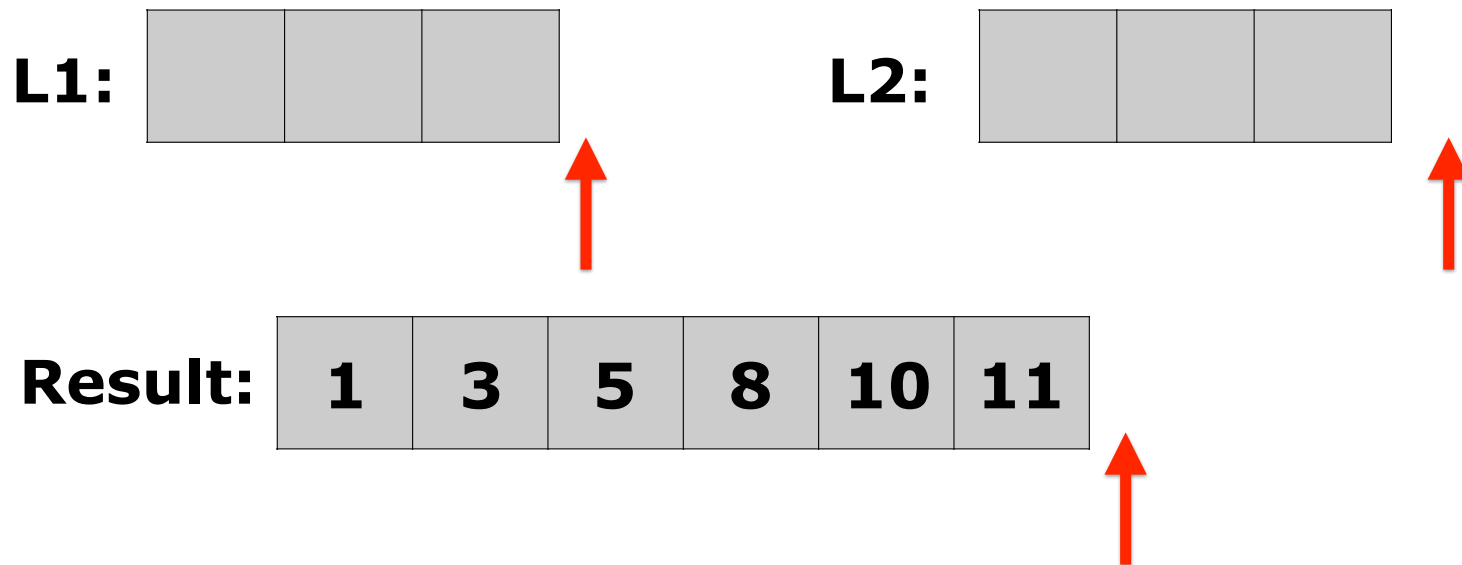
Merge Sort

- 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: Full Example

99	6	86	15	58	35	86	4	0
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------



Merge Sort: Full Example

99	6	86	15	58	35	86	4	0
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

99	6	86	15
-----------	----------	-----------	-----------

58	35	86	4	0
-----------	-----------	-----------	----------	----------



Merge Sort: Full Example

99	6	86	15	58	35	86	4	0
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

99	6	86	15
-----------	----------	-----------	-----------

58	35	86	4	0
-----------	-----------	-----------	----------	----------

99	6
-----------	----------

86	15
-----------	-----------

58	35
-----------	-----------

86	4	0
-----------	----------	----------



Merge Sort: Full Example

99	6	86	15	58	35	86	4	0
----	---	----	----	----	----	----	---	---

99	6	86	15
----	---	----	----

58	35	86	4	0
----	----	----	---	---

99	6
----	---

86	15
----	----

58	35
----	----

86	4	0
----	---	---

99	6
----	---

86	15
----	----

58	35
----	----

86

4	0
---	---



Merge Sort: Full Example

99	6	86	15	58	35	86	4	0
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

99	6	86	15
-----------	----------	-----------	-----------

58	35	86	4	0
-----------	-----------	-----------	----------	----------

99	6
-----------	----------

86	15
-----------	-----------

58	35
-----------	-----------

86	4	0
-----------	----------	----------

99	6
-----------	----------

86	15
-----------	-----------

58	35
-----------	-----------

86

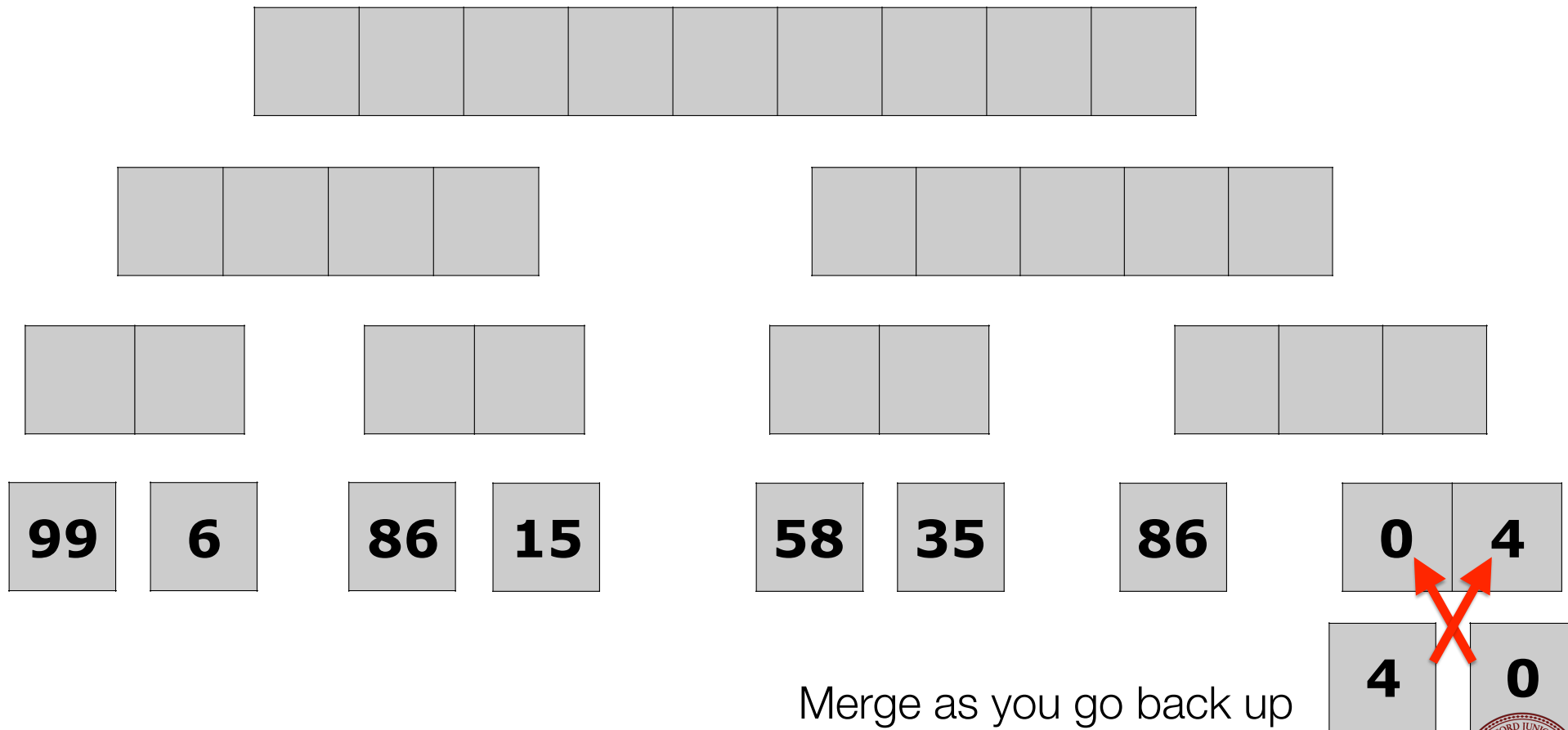
4	0
----------	----------

4

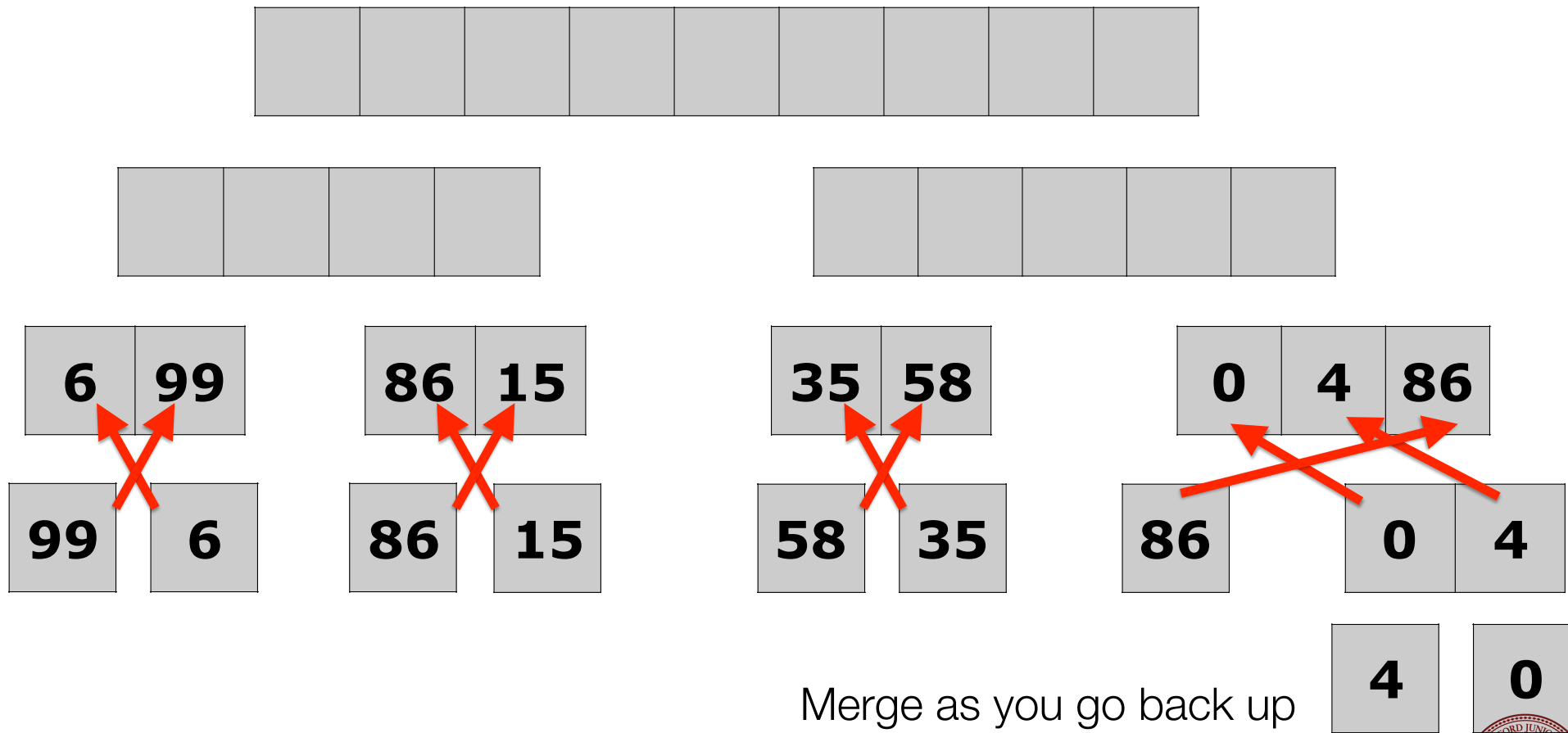
0



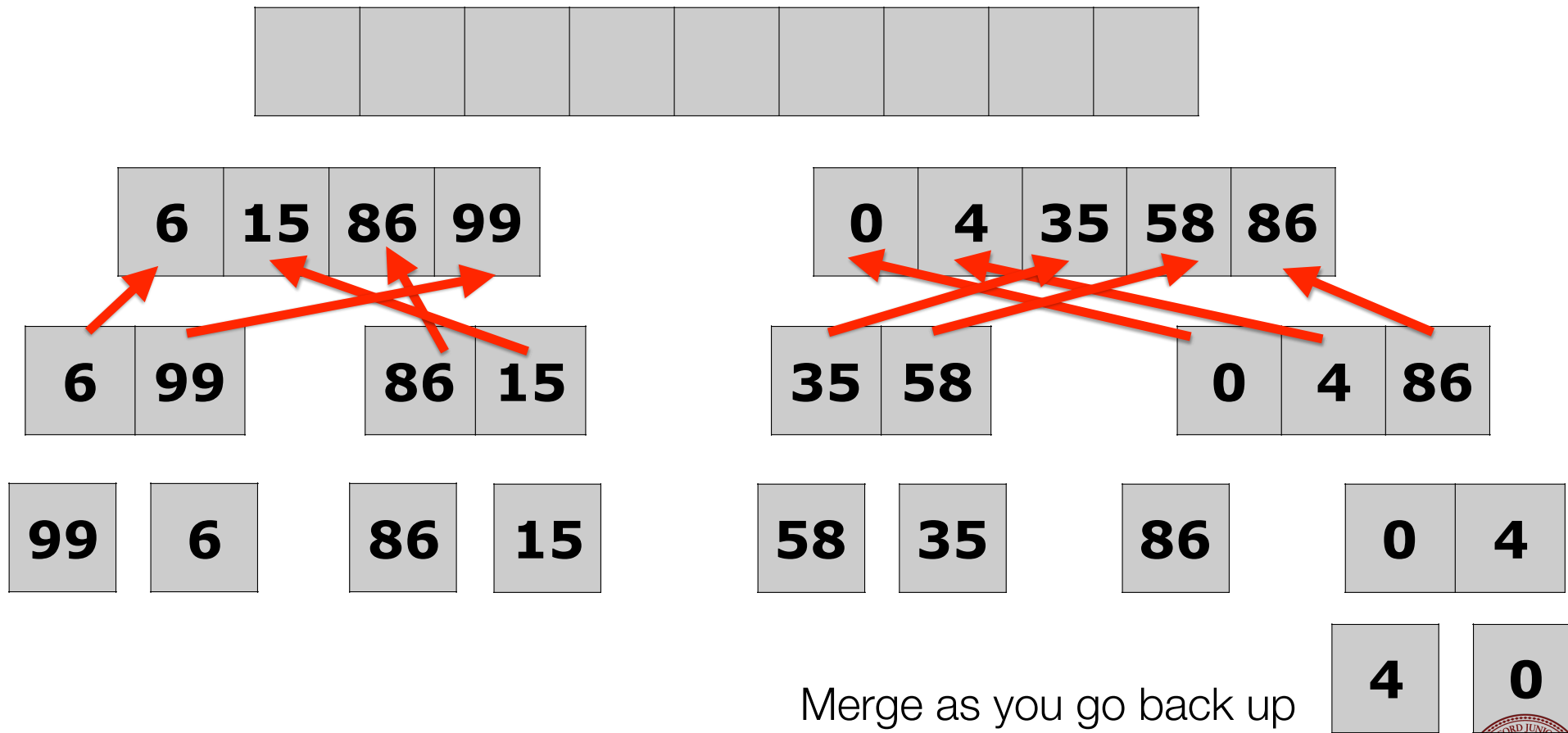
Merge Sort: Full Example



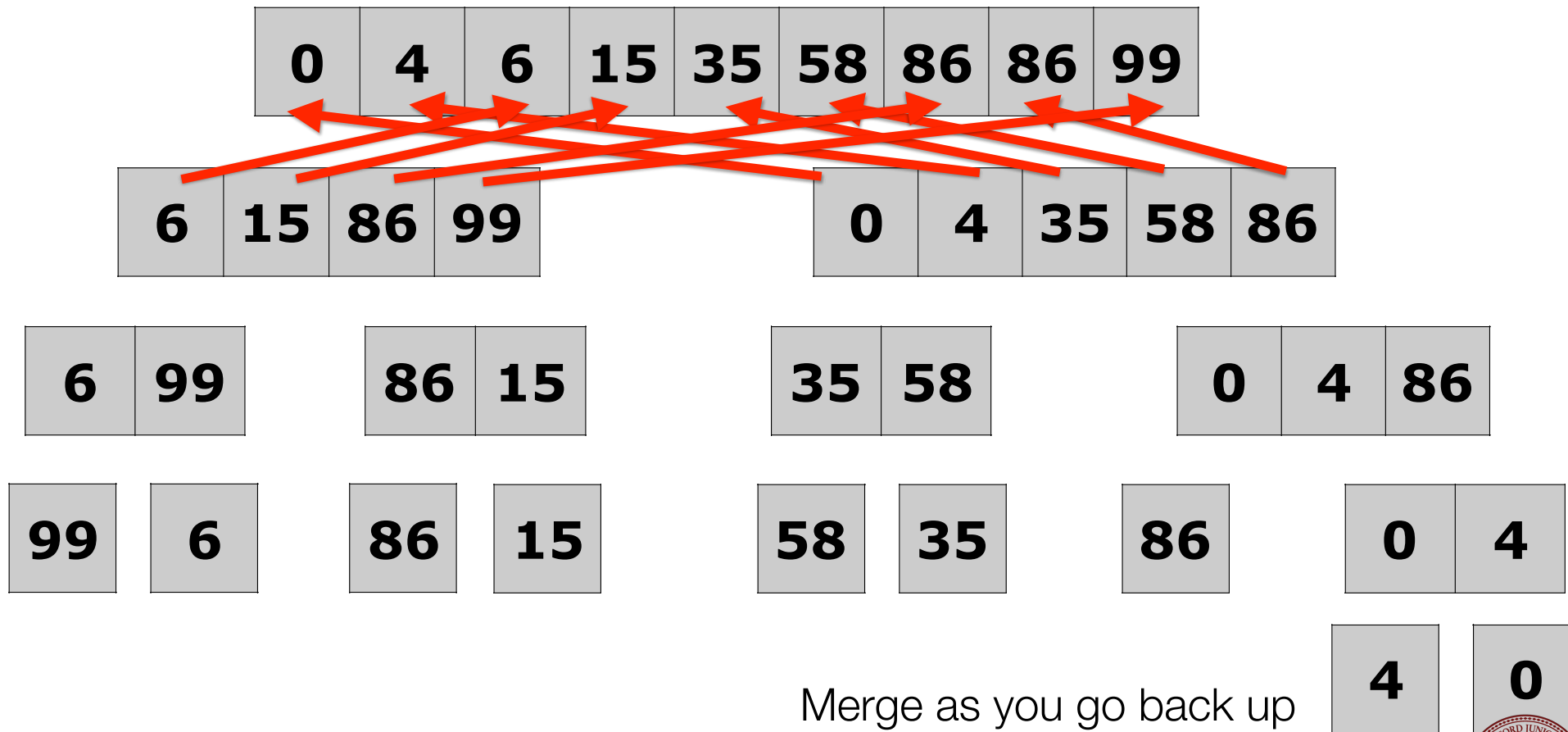
Merge Sort: Full Example



Merge Sort: Full Example



Merge Sort: Full Example



Merge Sort: Space Complexity

0	4	6	15	35	58	86	86	99
----------	----------	----------	-----------	-----------	-----------	-----------	-----------	-----------

- 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

0	4	6	15	35	58	86	86	99
----------	----------	----------	-----------	-----------	-----------	-----------	-----------	-----------

- 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.



Merge Sort Code (Recursive!)

```
// Rearranges the elements of v into sorted order using
// the merge sort algorithm.
void mergeSort(Vector<int>& v) {
    if (v.size() >= 2) {
        // split vector into two halves
        Vector<int> left;
        for (int i = 0; i < v.size()/2; i++) {
            left += v[i];
        }
        Vector<int> right;
        for (int i = v.size()/2; i < v.size(); i++) {
            right += v[i];
        }
        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);
        // merge the sorted halves into a sorted whole
        v.clear();
        merge(v, left, right);
    }
}
```



Merge Halves Code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
void merge(Vector<int>& result, Vector<int>& left, Vector<int>& right) {
    int i1 = 0;    // index into left side
    int i2 = 0;    // index into right side
    for (int i = 0; i < left.size() + right.size(); i++) {
        if (i2 >= right.size()
            || (i1 < left.size()
                && left[i1] <= right[i2])) {
            // take from left
            result += left[i1];
            i1++;
        } else {
            // take from right
            result += right[i2];
            i2++;
        }
    }
}
```



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

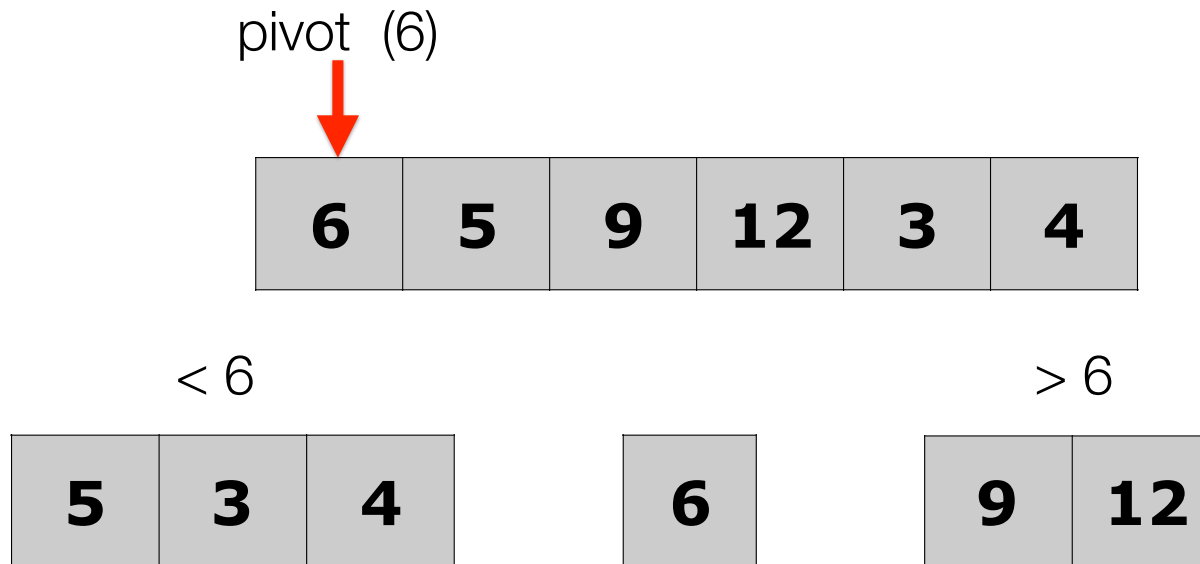
pivot (6)



6	5	9	12	3	4
----------	----------	----------	-----------	----------	----------



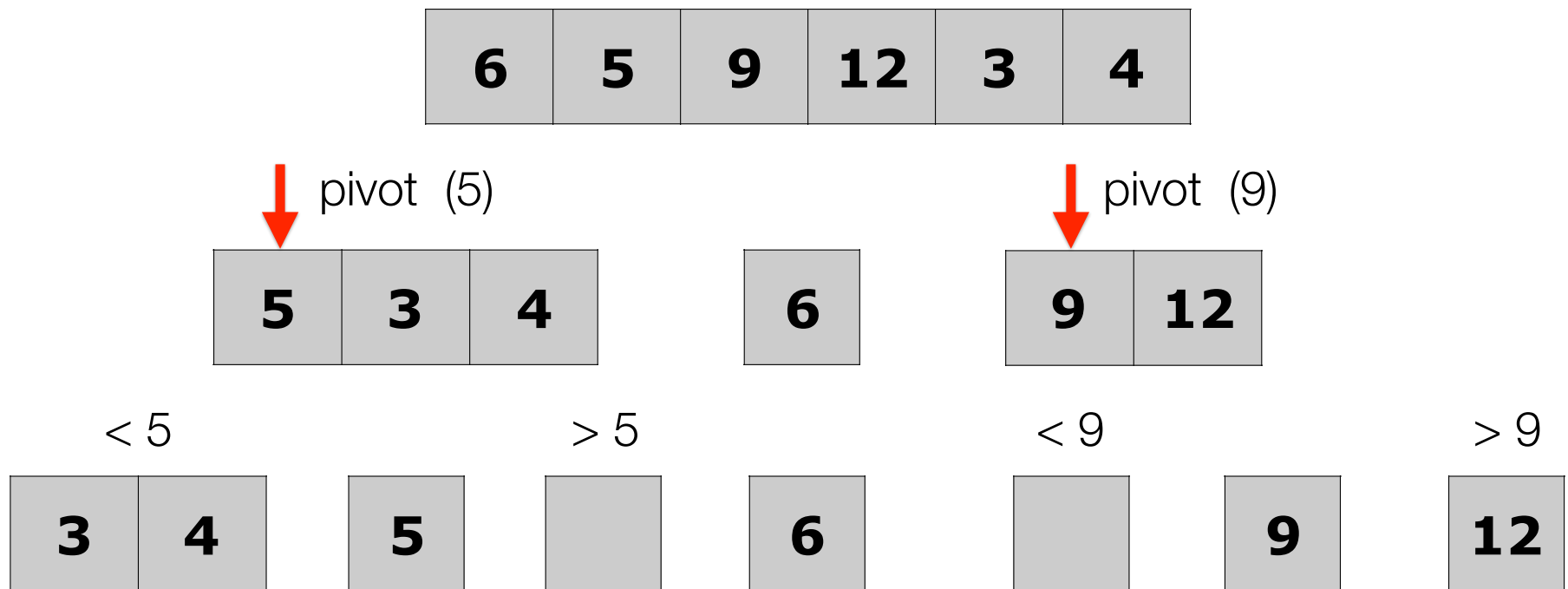
Quicksort Algorithm: Naive



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.



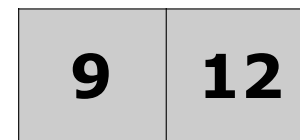
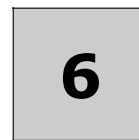
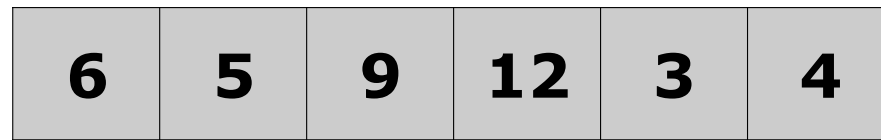
Quicksort Algorithm: Naive



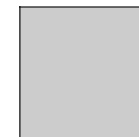
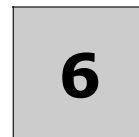
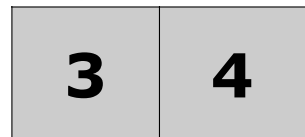
Keep partitioning the sub-lists



Quicksort Algorithm: Naive

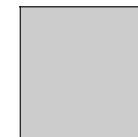
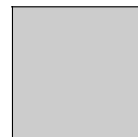


↓ pivot (3)

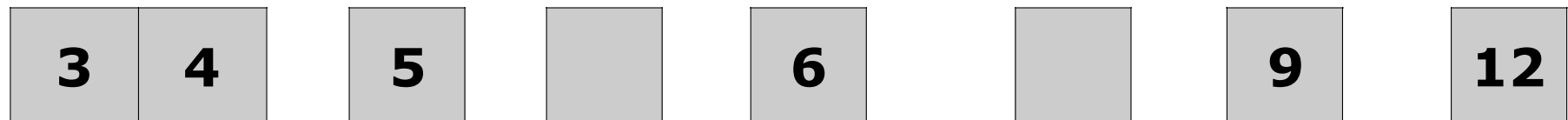
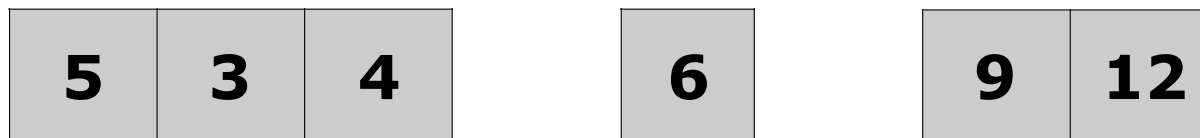
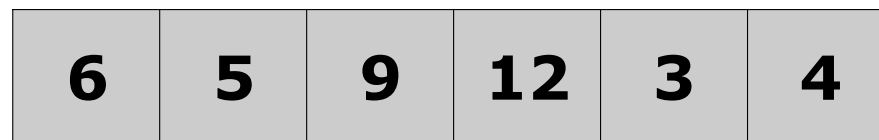


< 3

> 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	5	9	12	3	4



pivot (6)

In-place, recursive algorithm:

```
int quickSort(vector<int> &v, int leftIndex, int rightIndex);
```

- Pick your pivot, and swap it with the end element.
- Traverse the list from the beginning (left) forwards until the value should be to the *right* of the pivot.
- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot.
- Swap the pivot (now at the end) with the element where the left/right cross.

This is best described with a detailed example...



Quicksort Algorithm: In-Place

0	1	2	3	4	5
6	5	9	12	3	4

↑
pivot (6)

- Pick your pivot, and swap it with the end element.

`quickSort(vector, 0, 5)`



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	9	12	3	6



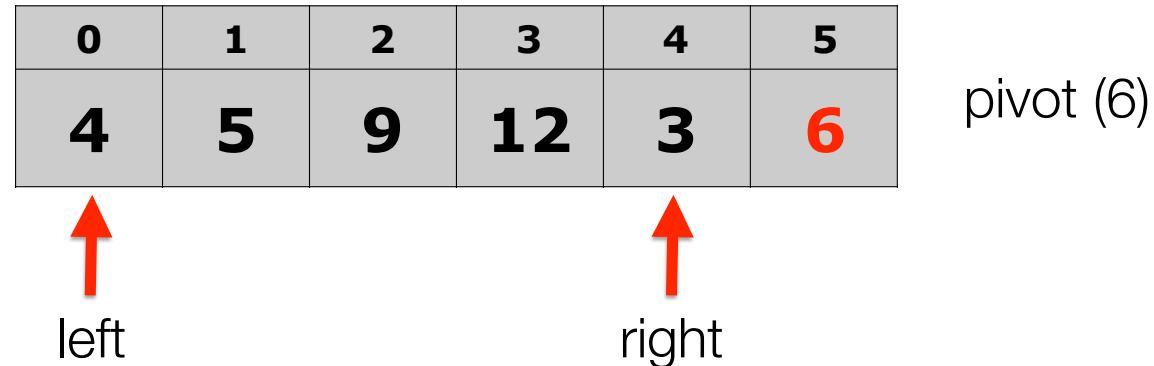
pivot (6)

- Pick your pivot, and swap it with the end element.

`quickSort(vector, 0, 5)`



Quicksort Algorithm: In-Place



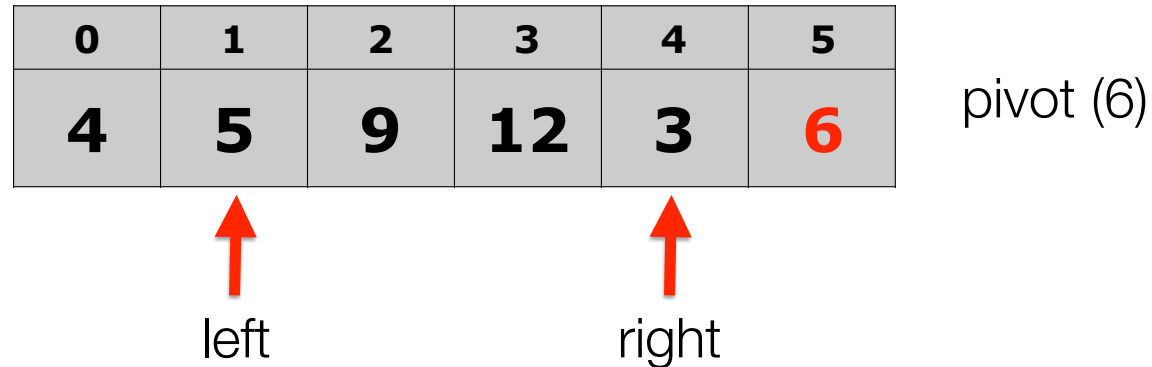
Choose the "left" / "right" indices to be at the start (after the pivot) / end of your vector.

Traverse the list from the beginning (left) forwards until the value should be to the *right* of the pivot.

```
quickSort(vector, 0, 5)
```



Quicksort Algorithm: In-Place



Traverse the list from the beginning (left) forwards until the value should be to the *right* of the pivot.

```
quickSort(vector, 0, 5)
```



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	9	12	3	6

pivot (6)

9 should be to the right
of the pivot



- Traverse the list from the end (right) backwards until the value should be to the *left* of the pivot.

`quickSort(vector, 0, 5)`



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	9	12	3	6

pivot (6)



left



right

3 should be to the left
of the pivot

- The left element and the right element are out of order, so we swap them, and move our left/right indices.

```
quickSort(vector, 0, 5)
```



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	3	12	9	6

pivot (6)



3 should be to the left
of the pivot

- The left element and the right element are out of order, so we swap them, and move our left/right indices.

`quickSort(vector, 0, 5)`



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	3	12	9	6

pivot (6)

↑
left
↑
right

3 should be to the left
of the pivot

When the left and right cross each other, we return the index of the left/right, and then swap the left and the pivot.

`quickSort(vector, 0, 5)`



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	3	6	9	12

pivot (6)



3 should be to the left
of the pivot

When the left and right cross each other, we return the index of the left/right, and then swap the left and the pivot.

return left;

Notice that we have partitioned correctly: all the elements to the left of the pivot are less than the pivot, and all the elements to the right are greater than the pivot.



Quicksort Algorithm: In-Place

0	1	2	3	4	5
4	5	3	6	9	12

Recursively call quickSort() on the two new partitions
The original pivot is now in the proper place and does not need to be re-sorted.

quickSort(vector, 0, 2)

quickSort(vector, 4, 5)



Quicksort Algorithm: Choosing the Pivot

0	1	2	3	4	5
4	5	3	6	9	12

- One interesting issue with quicksort is the decision about choosing the pivot.
- If the left-most element is always chosen as the pivot, already-sorted arrays will have $O(n^2)$ behavior (why?)
- Therefore, choosing a pivot that is random works well, or choosing the middle item as the pivot.



Quicksort Algorithm: Repeated Elements

0	1	2	3	4	5
5	5	4	6	5	5

- Repeated elements also cause quicksort to slow down.
- If the whole list was the same value, each recursion would cause all elements to go into one partition, which degrades to $O(n^2)$
- The solution is to separate the values into three groups: values less than the pivot, values equal to the pivot, and values greater than the pivot (sometimes called Quick3)



Quicksort Algorithm: Big-O

0	1	2	3	4	5
3	5	4	6	12	9

- 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)



Quicksort In-place Code

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

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



Quicksort In-place Code: Helper Function

```
void quickSortHelper(Vector<int>& v, int min, int max) {
    if (min >= max) { // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = v[min];
    swap(v, min, max); // move pivot to end

    // partition the two sides of the array
    int middle = partition(v, min, max - 1, pivot);

    swap(v, middle, max); // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSortHelper(v, min, middle - 1);
    quickSortHelper(v, middle + 1, max);
}
```



Quicksort In-place Code: Partition Function

```
// Partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
int partition(Vector<int>& v, int left, int right, int pivot) {
    while (left <= right) {
        // move index markers left, right toward center
        // until we find a pair of out-of-order elements
        while (left <= right && v[left] < pivot) {
            left++;
        }
        while (left <= right && v[right] > pivot) {
            right--;
        }

        if (left <= right) {
            swap(v, left++, right--);
        }
    }
    return left;
}
```



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 (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
- 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
- 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

