CS 106B
Lecture 11: Sorting

Friday, October 21, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Section 10.2

Today's Topics

- Logistics
  - We will have a midterm review, TBA
  - Midterm materials (old exams, study guides, etc.) will come out next week.
- Throwing a string error
- Recursive != exponential computational complexity
- 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

Throwing an Exception

- For the Serpinski Triangle option in MetaAcademy assignment says, "*If the order passed is negative, your function should throw a string exception*."
- What does it mean to "throw a string exception?"
- An "exception" is your program's way of pulling the fire alarm — something drastic happened, and your program does not like it. In fact, if an exception is not "handled" by the rest of your program, it crashes! It is not a particularly graceful way to handle bad input from the user.
- If you were going to use your Serpinski function in a program you wrote, you would want to check the input before calling the function — in other words, make sure your program never creates a situation where a function needs to throw an exception.

Throwing an Exception

- To throw an exception:

```
throw("Illegal level: Serpinski level must be non-negative.");
```

- This will crash the program.
- You can "catch" exceptions that have been thrown by other functions, but that is beyond our scope — see here for details: https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

(that is a creepy hand)

Recursion != Exponential Computational Complexity

- In the previous lecture, we discussed the recursive fibonacci function:

```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

- This function happens to have exponential computational complexity because each recursive call has two additional recursions. Not all recursive functions are exponential. What is the complexity of the following?

```
long factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

- Answer: O(n)
- There is only one more recursive call for each additional n.

- For the midterm:
  - You should be able to determine basic Big-O for non-recursive functions, and we will cover more examples in detail.

Big-O Warmup

```
// What is the Big-O of the following?
void mysteryFunc(Vector<int>& v) {
    int n = v.size();
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < i; j++) {
            cout << "CS 106B Rocks (Trees?)" << endl;
        }
    }
}
```

How many times does the cout happen?

i	1	2	3	...	n
couts	1	2	3	...	n

total couts:	1+2+3+...+n

Big-O Warmup

total couts:	1+2+3+...+n

What is this function?

$$1 + 2 + 3 + \cdots + n = \frac{n \times (n+1)}{2}$$

$$= \frac{n^2 + n}{2} = \boxed{\frac{n^2}{2} + \frac{n}{2}}$$

$O(n^2)$

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

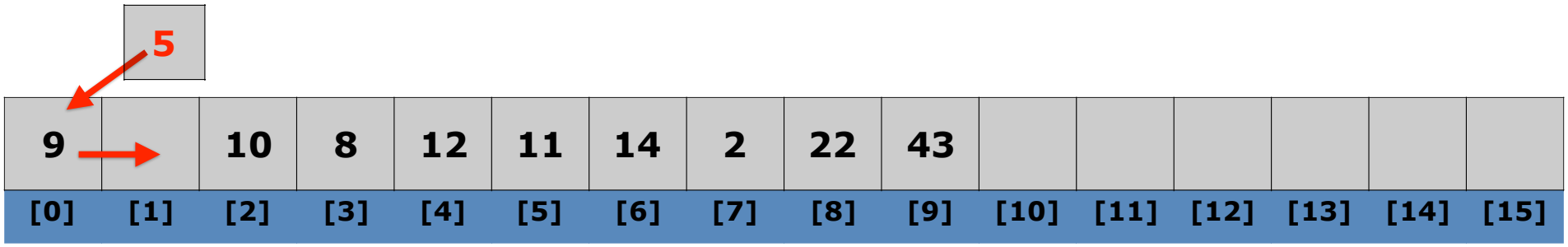
Insertion Sort

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

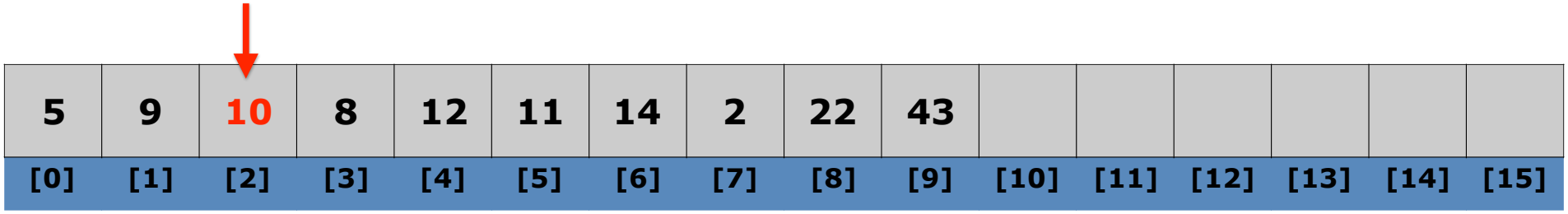


Algorithm:

- 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

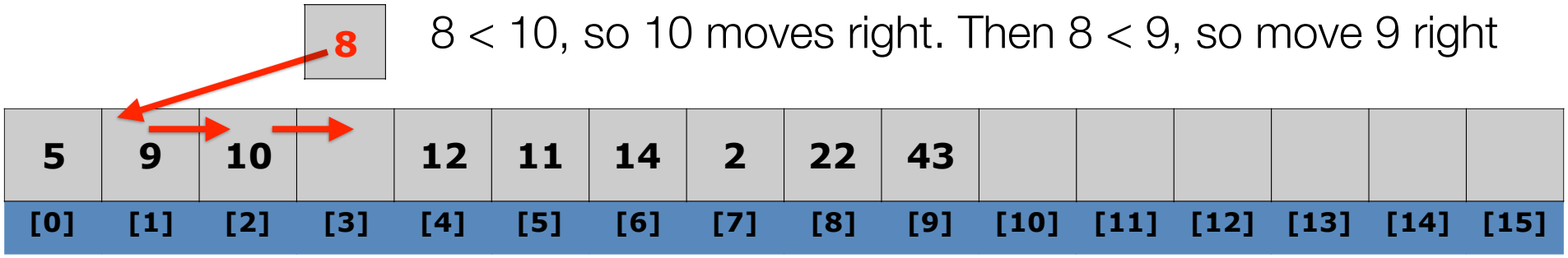
in place already (i.e., already bigger than 9)



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

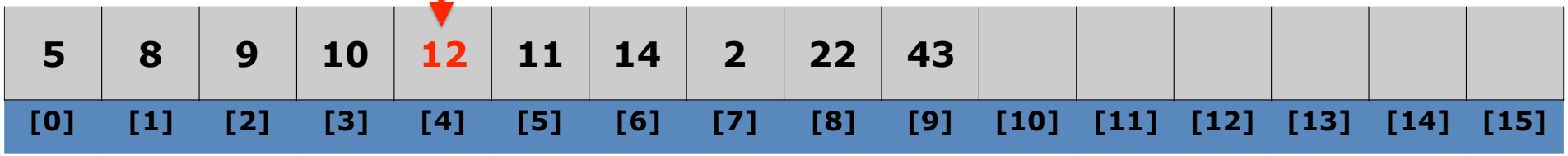


Algorithm:

- 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

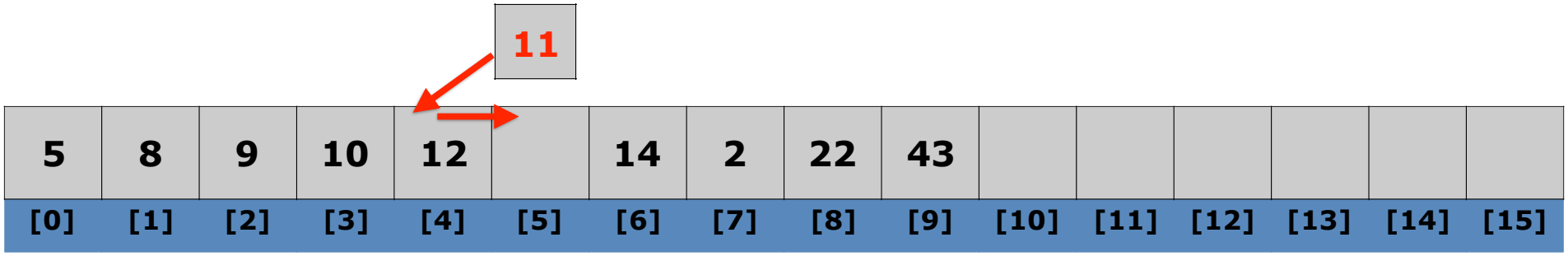
in place already (i.e., already bigger than 10)



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

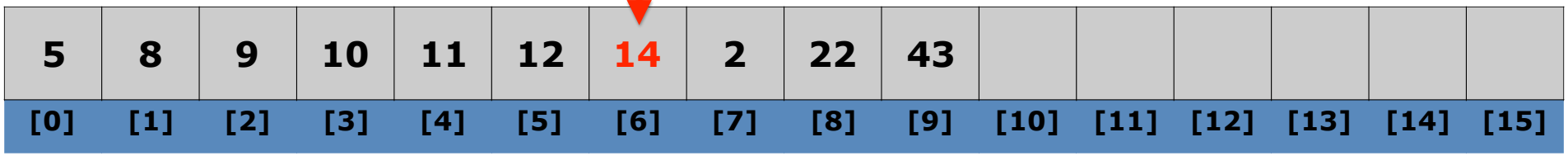


Algorithm:

- 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

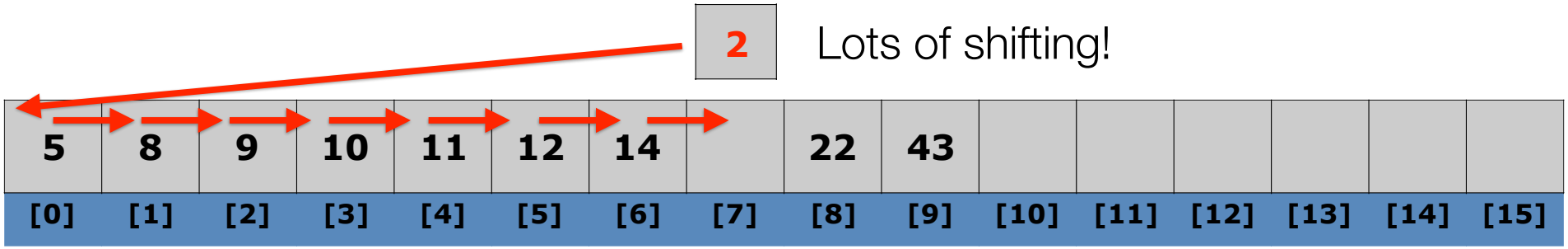
in place already (i.e., already bigger than 12)



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.

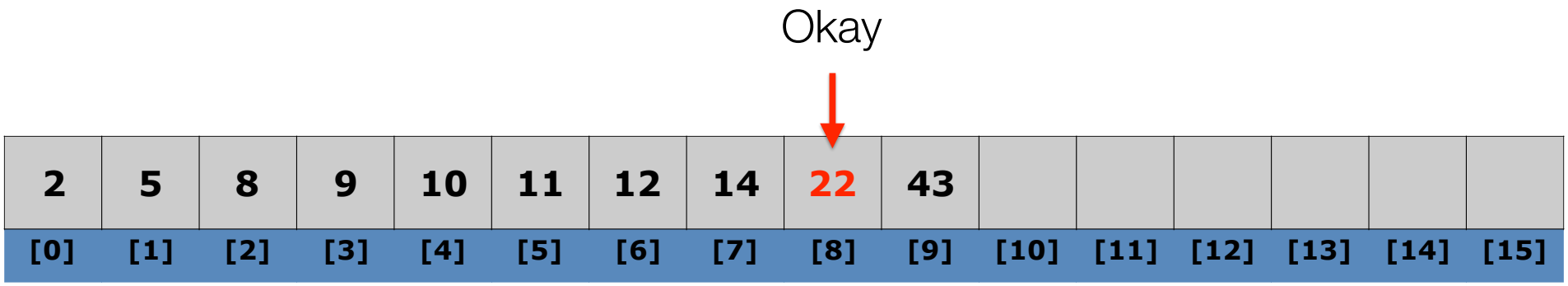
Insertion Sort



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.

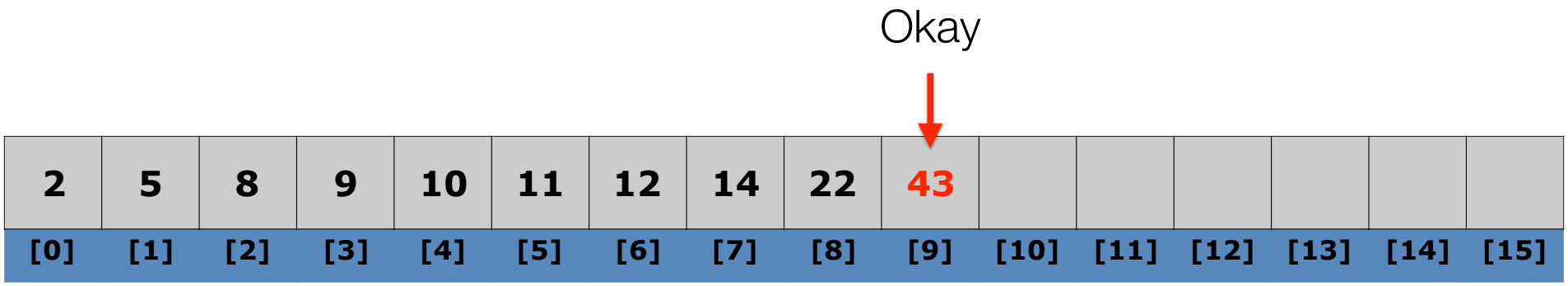
Insertion Sort



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



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

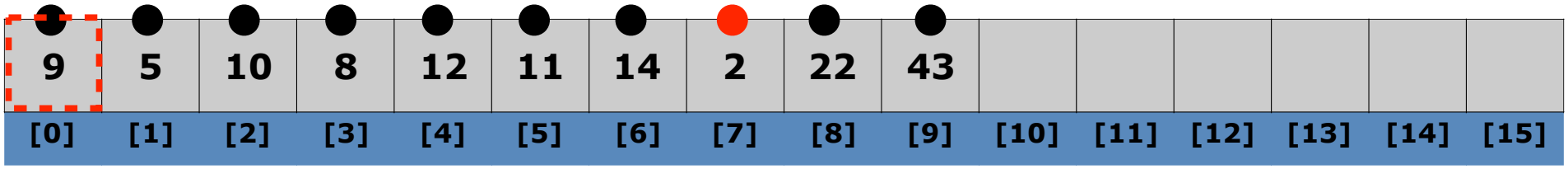
Selection Sort

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

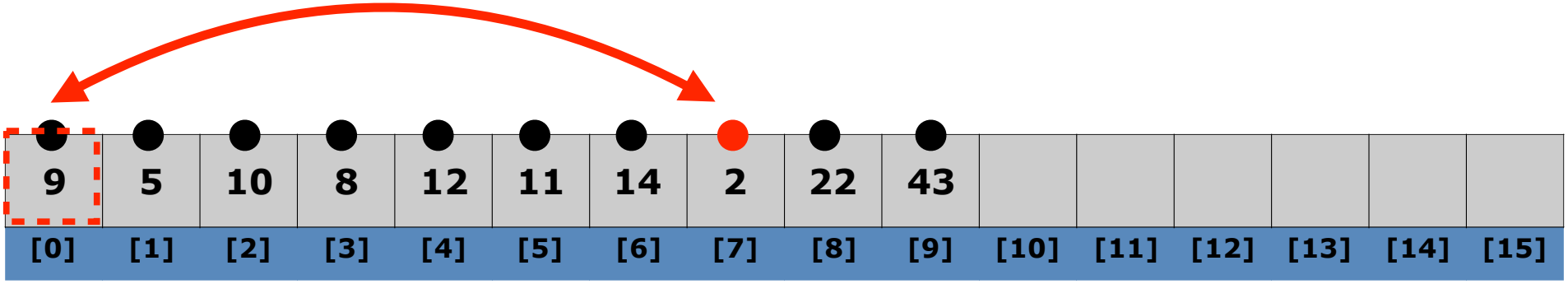
Selection Sort



- 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

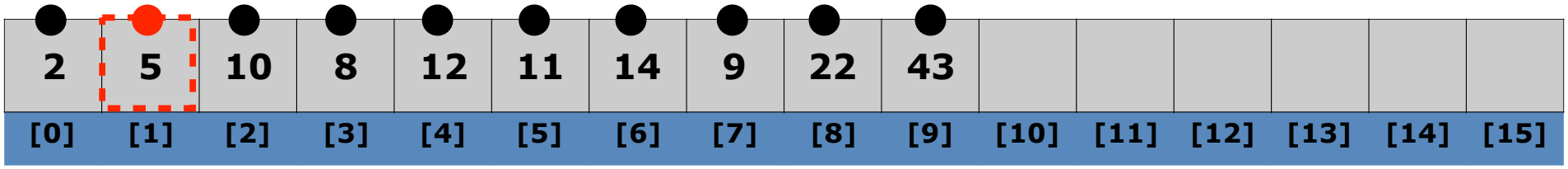


- Algorithm
  - Find the **smallest item in the list**, and exchange it with the left-most unsorted element.
  - Repeat the process from the first unsorted element.

- Selection sort is particularly slow, because it needs to go through **the entire list** each time to find the smallest item.

Selection Sort

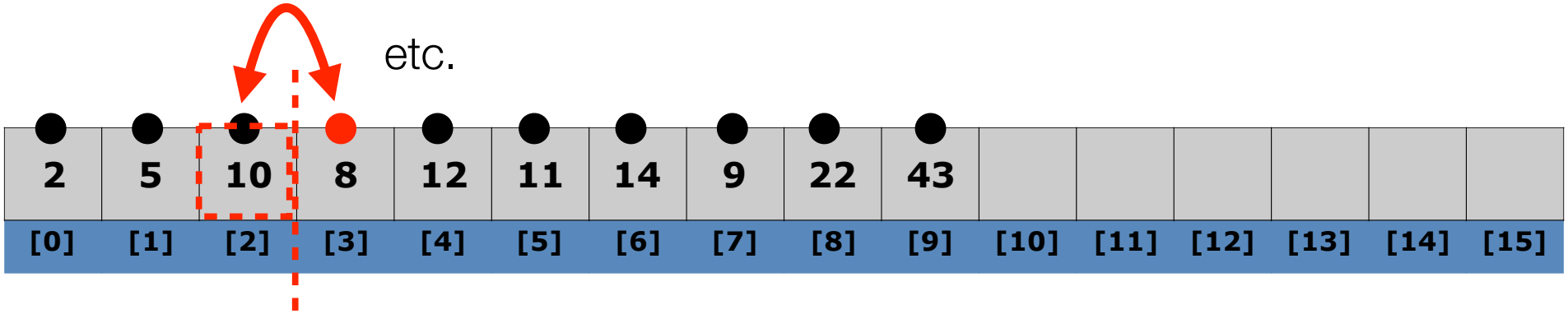
(no swap necessary)



- 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

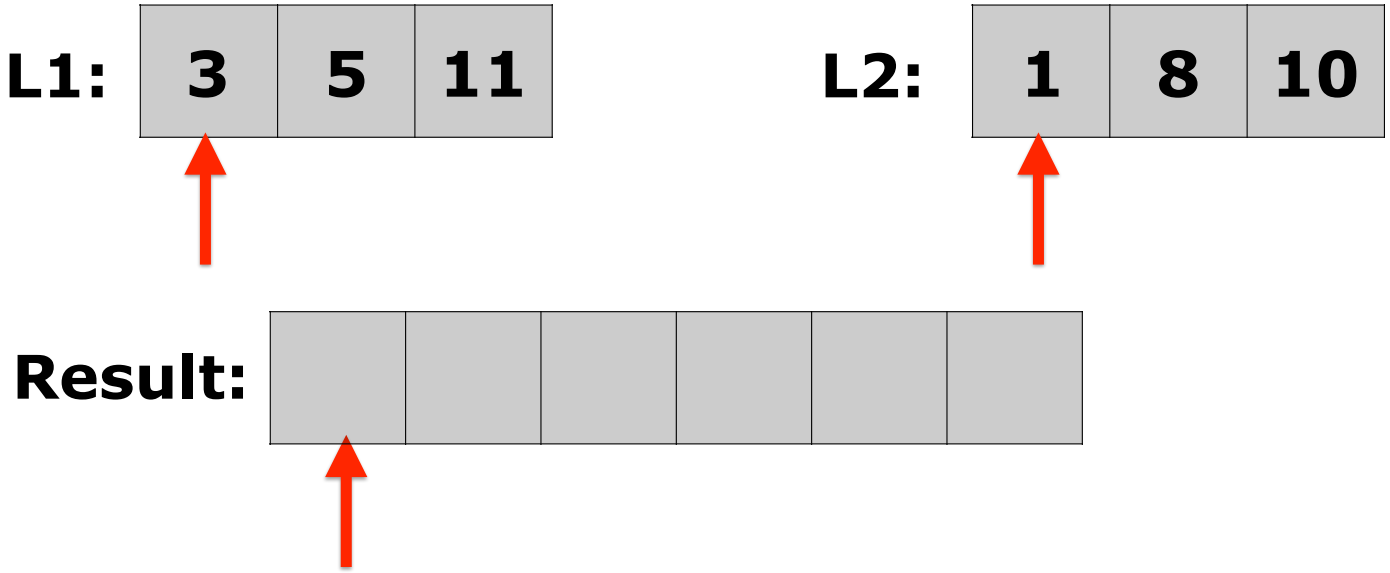
Quicksort

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

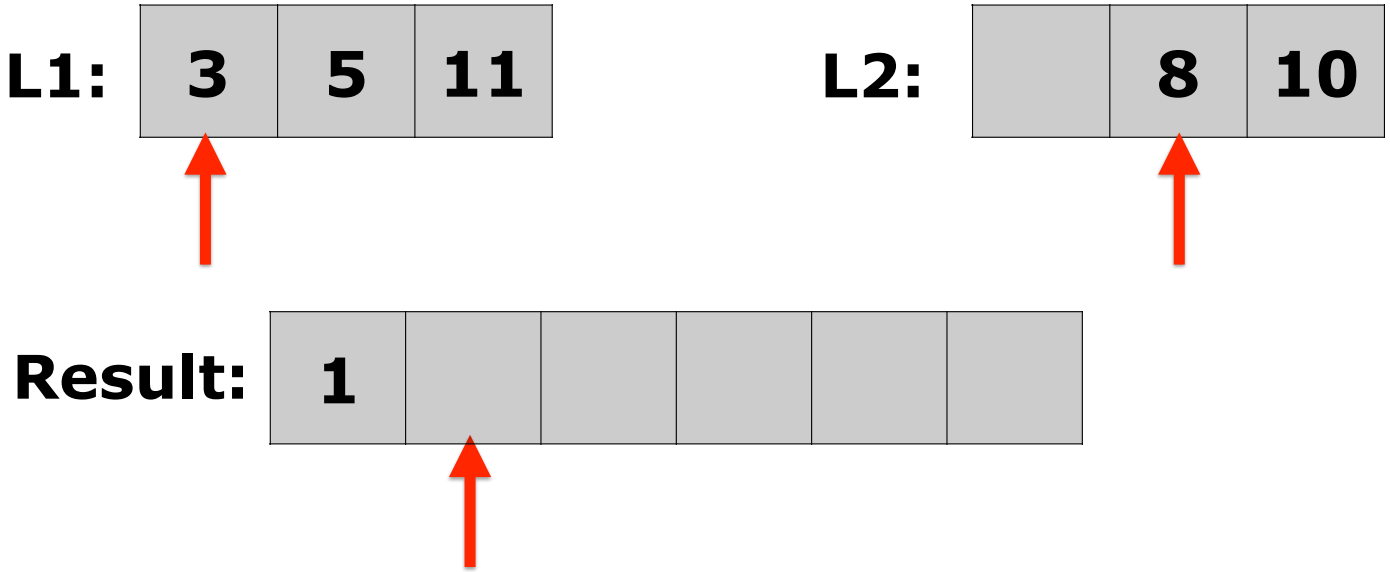
Merge Sort

- Merging two sorted lists is easy:



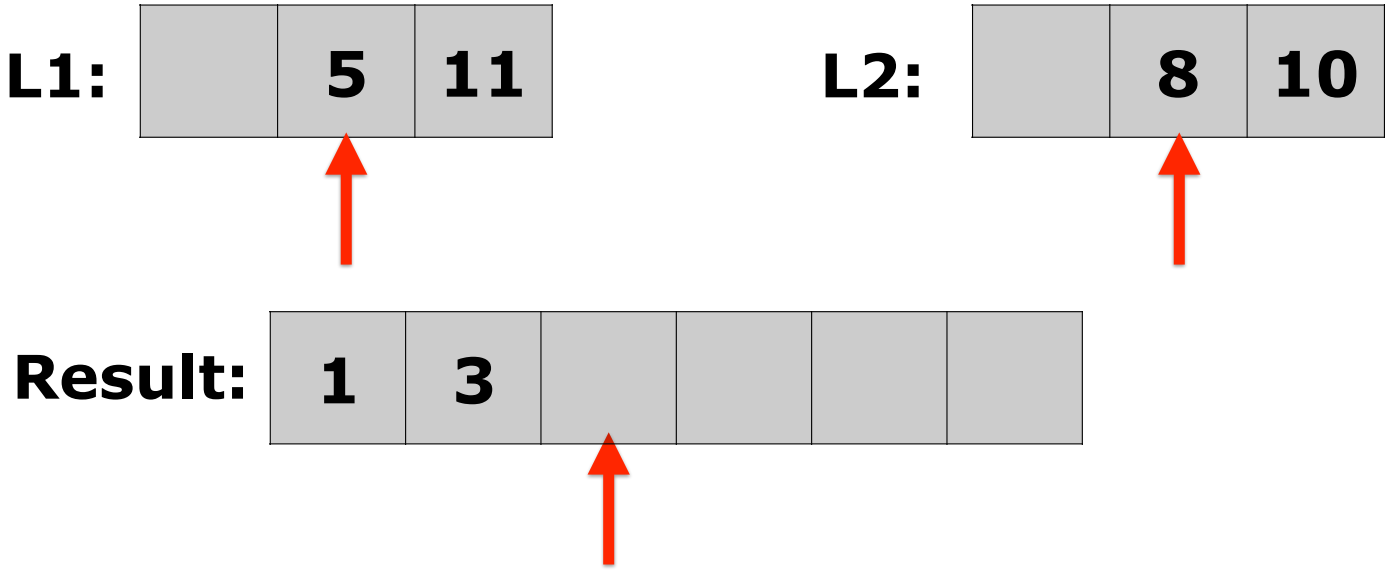
Merge Sort

- Merging two sorted lists is easy:



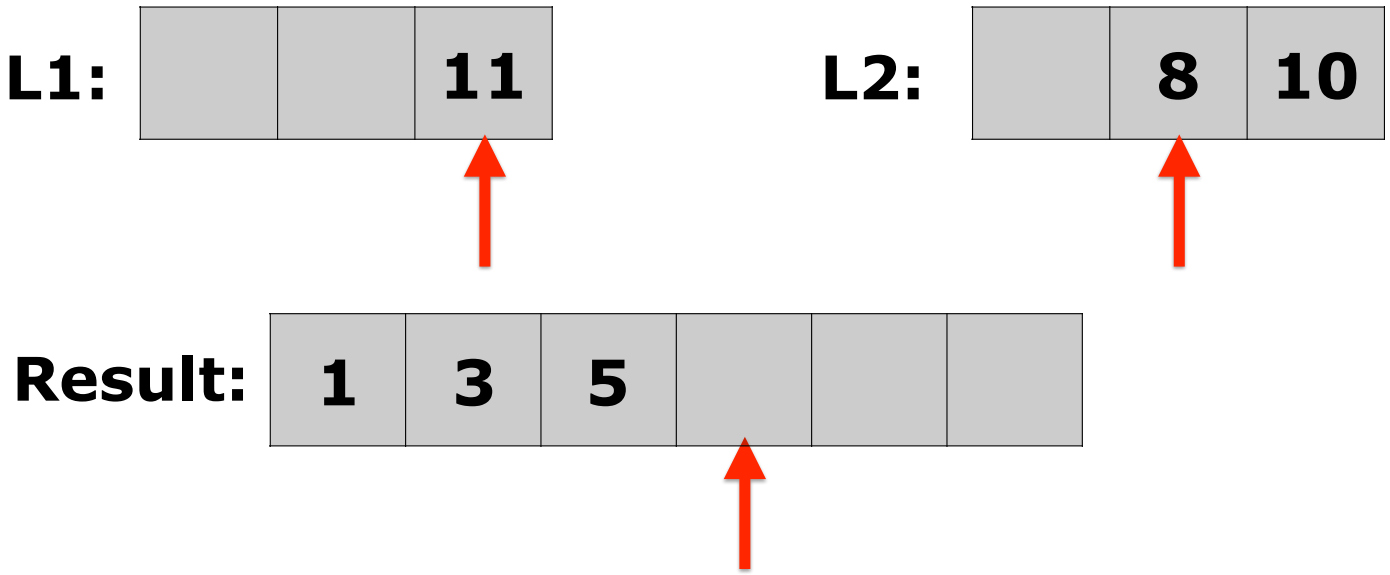
Merge Sort

- Merging two sorted lists is easy:



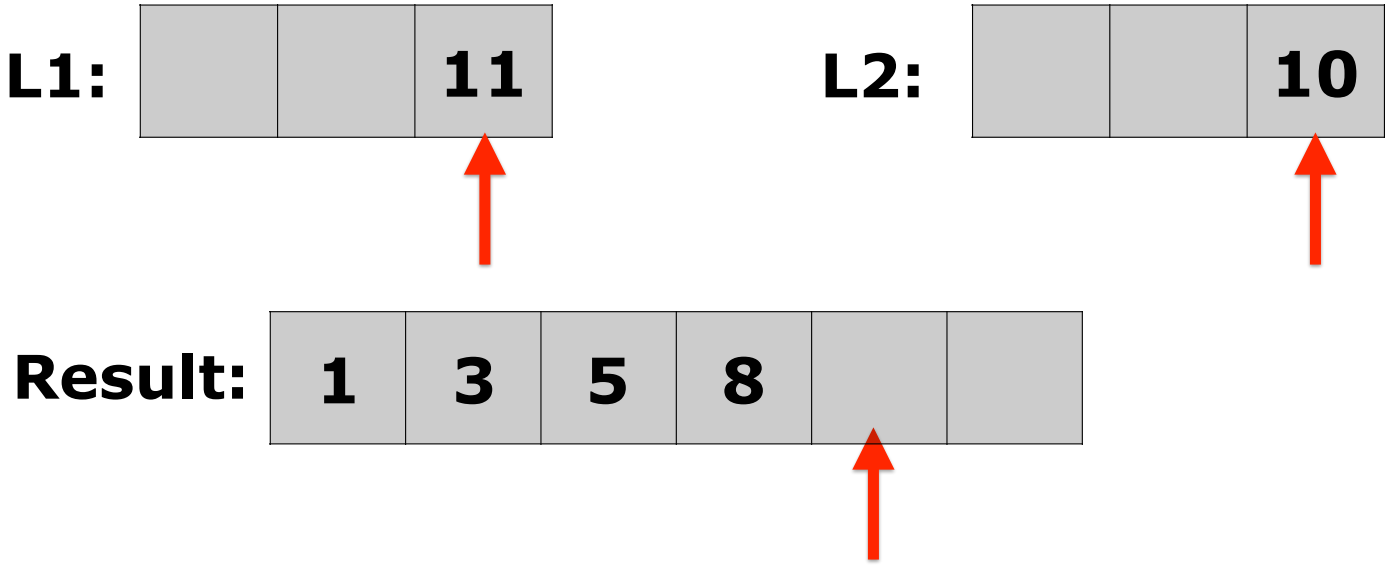
Merge Sort

- Merging two sorted lists is easy:



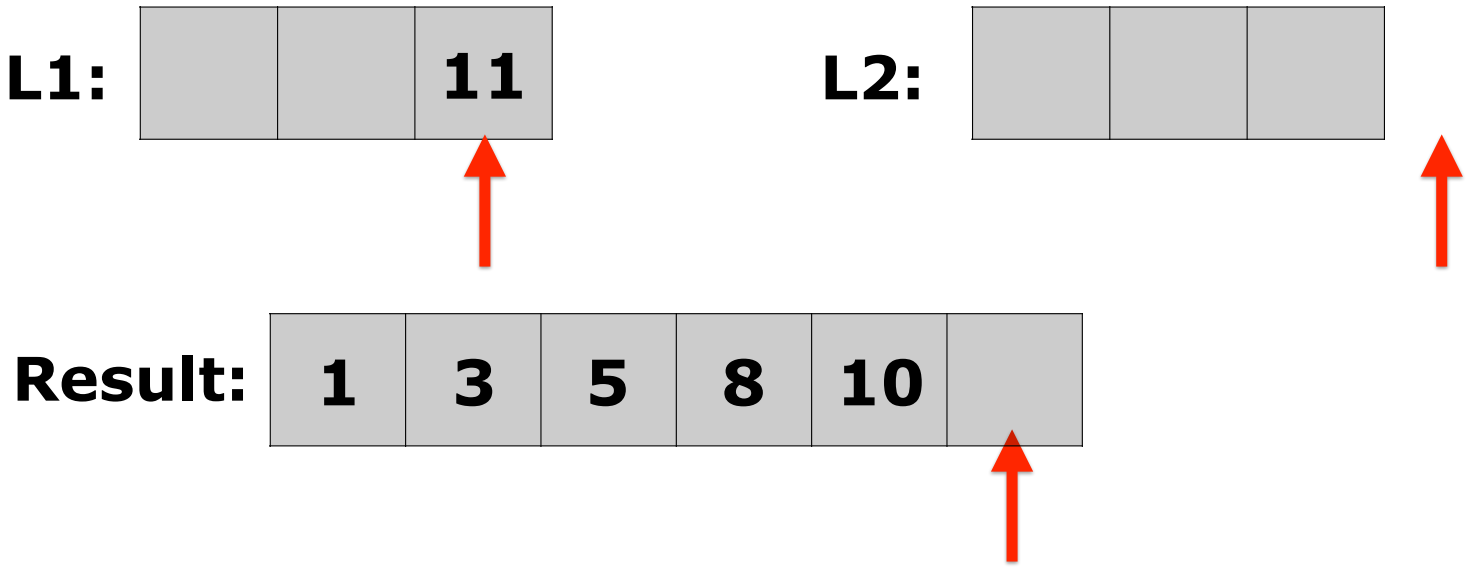
Merge Sort

- Merging two sorted lists is easy:



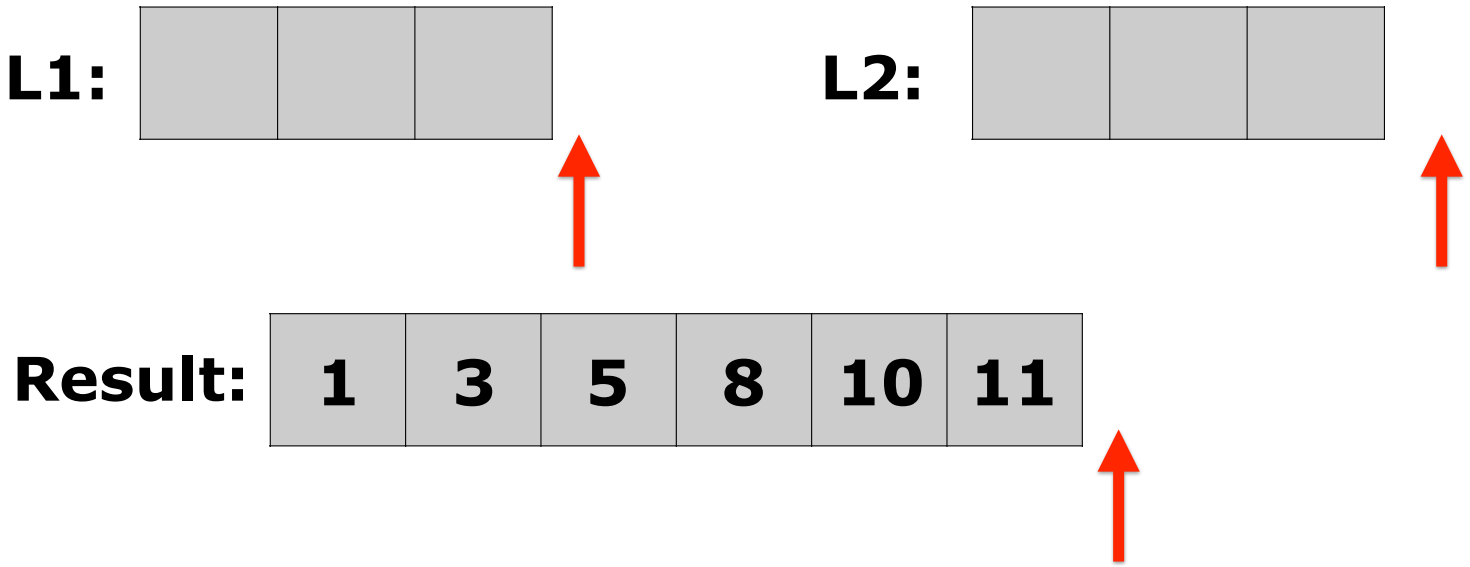
Merge Sort

- 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

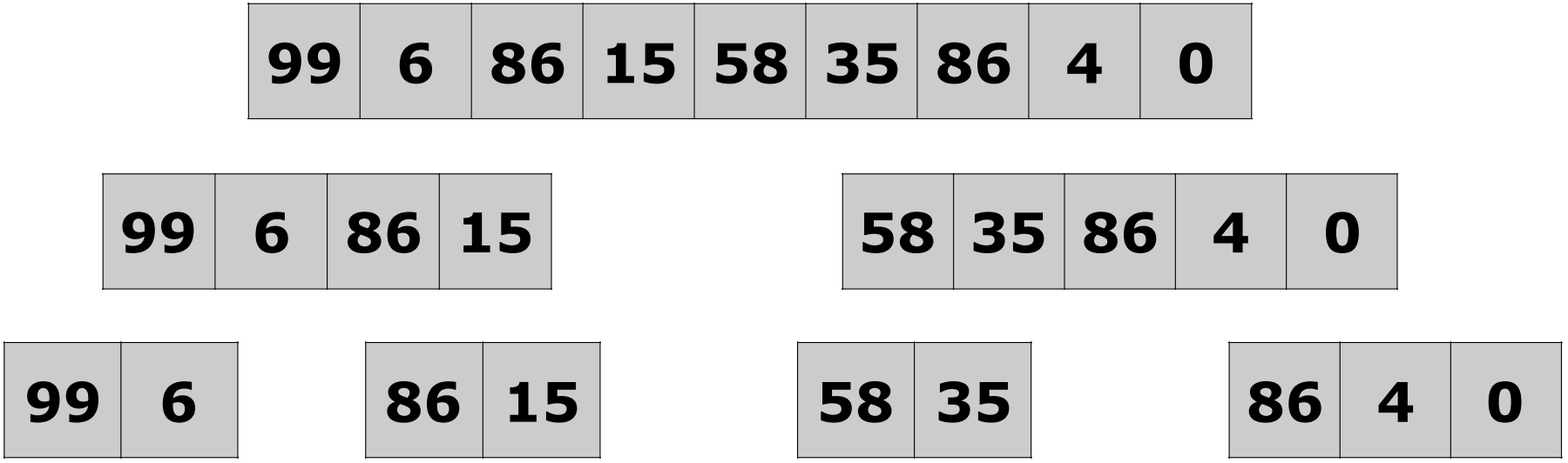
Merge Sort: Full Example

99	6	86	15	58	35	86	4	0

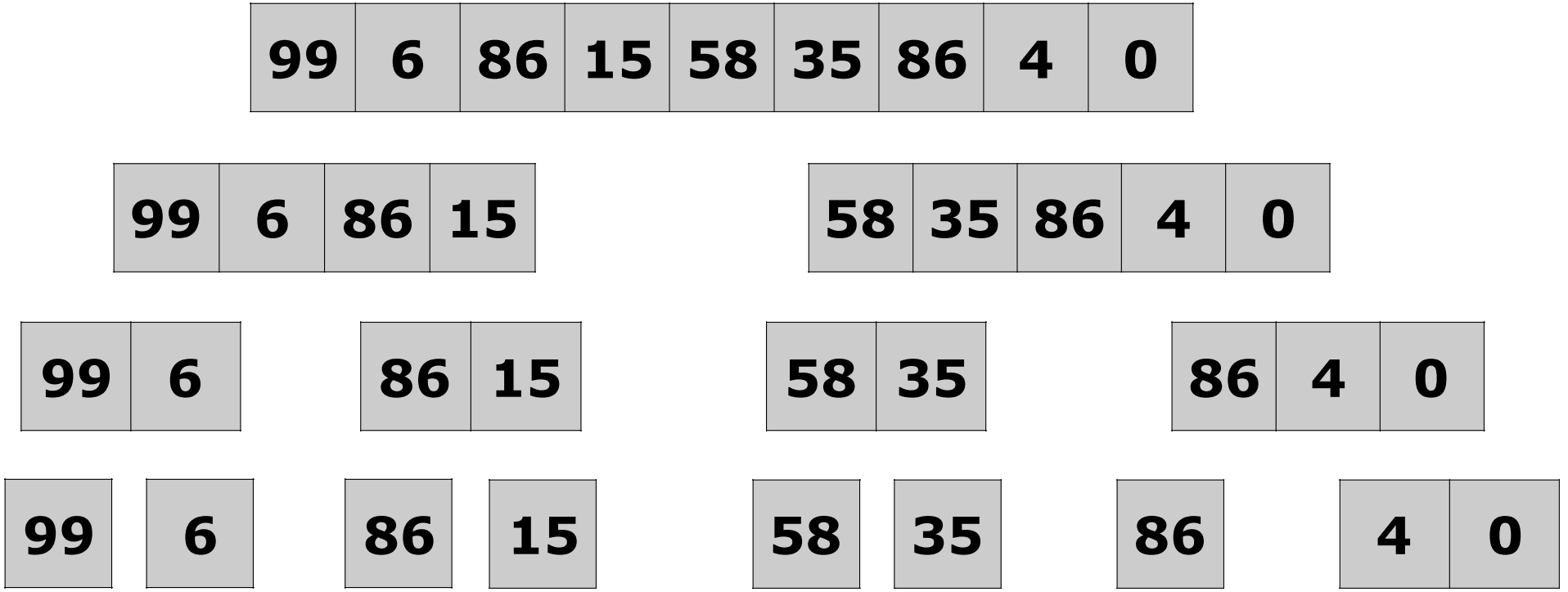
99	6	86	15

58	35	86	4	0

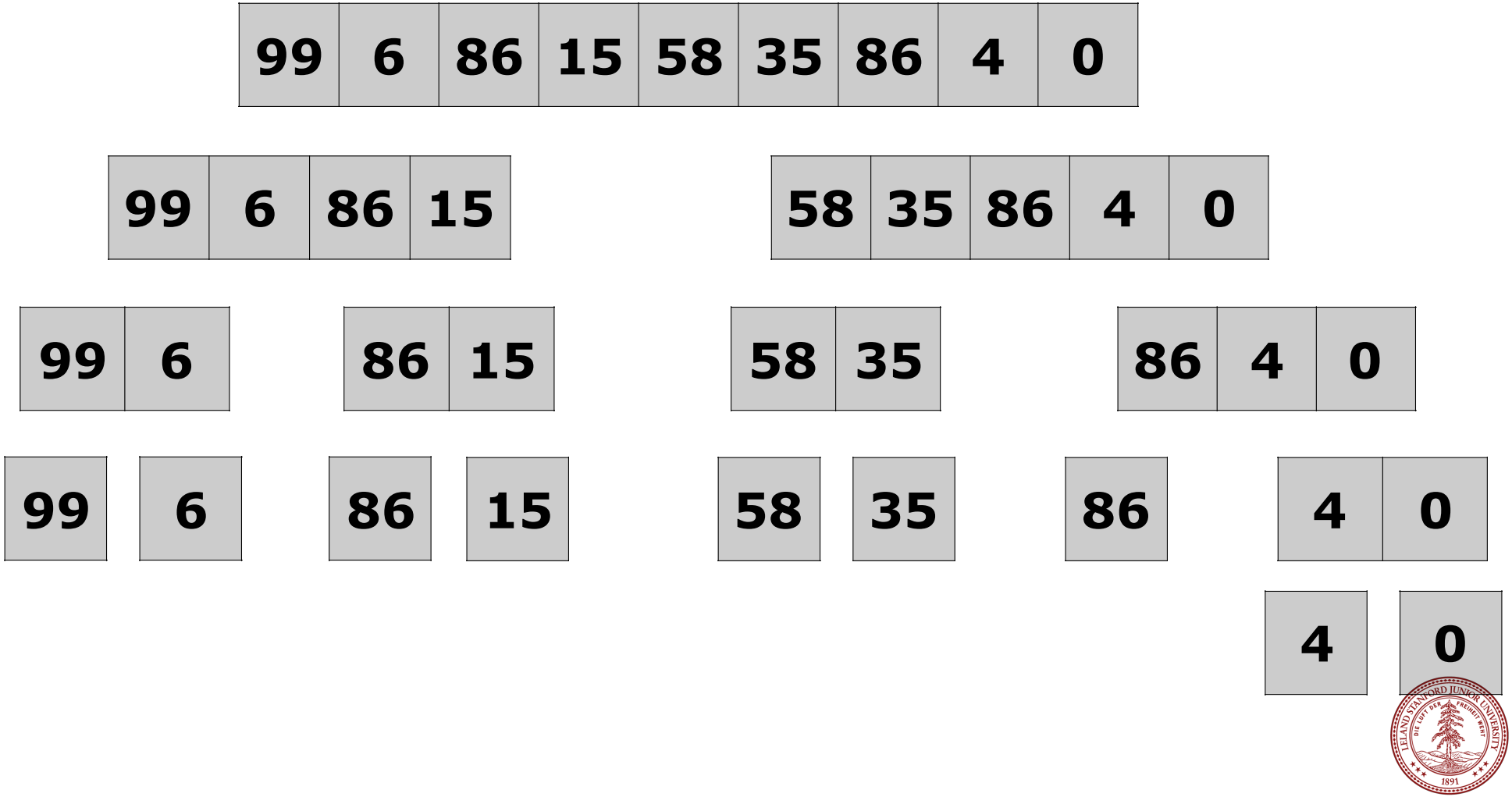
Merge Sort: Full Example



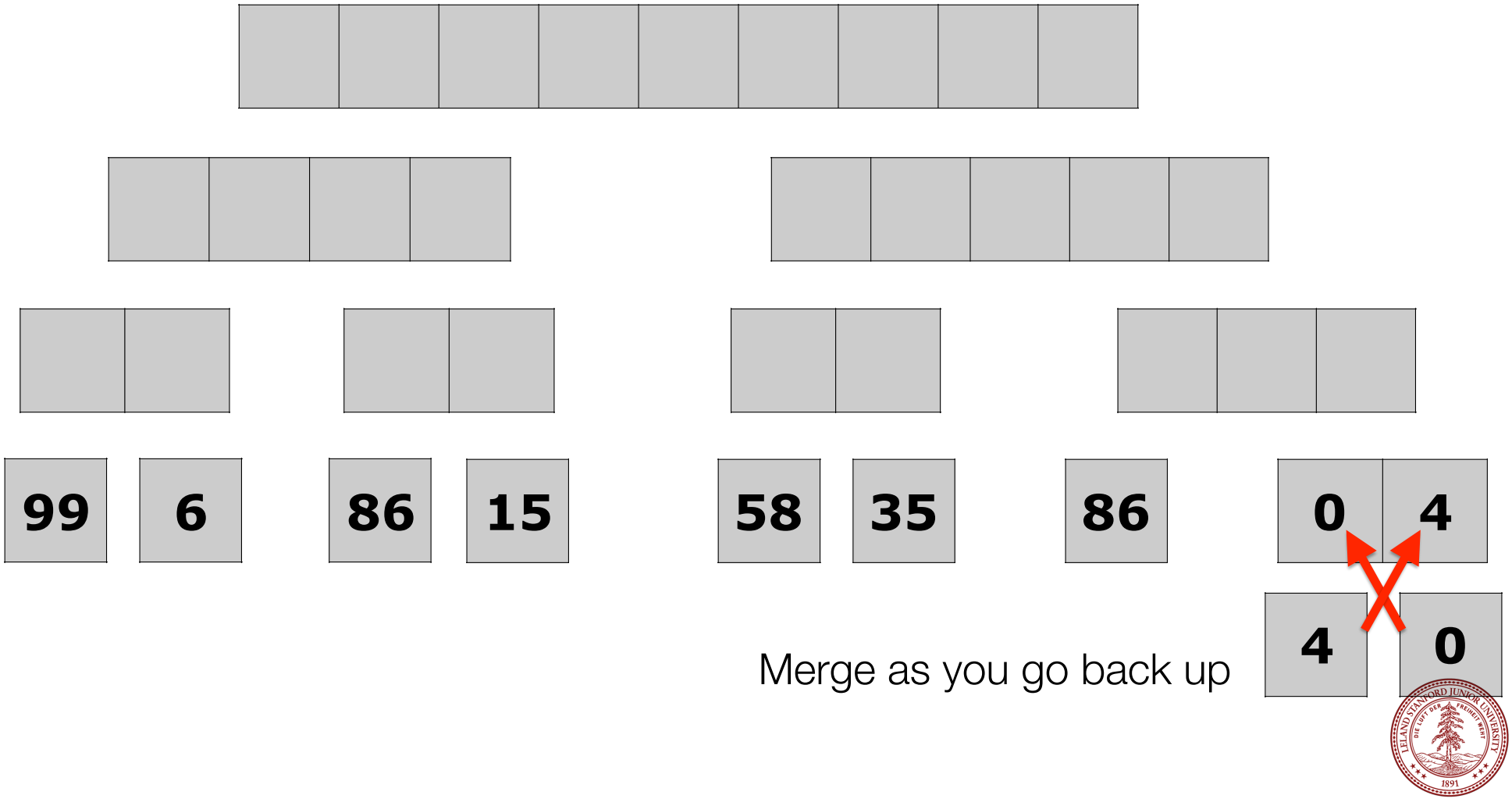
Merge Sort: Full Example



Merge Sort: Full Example

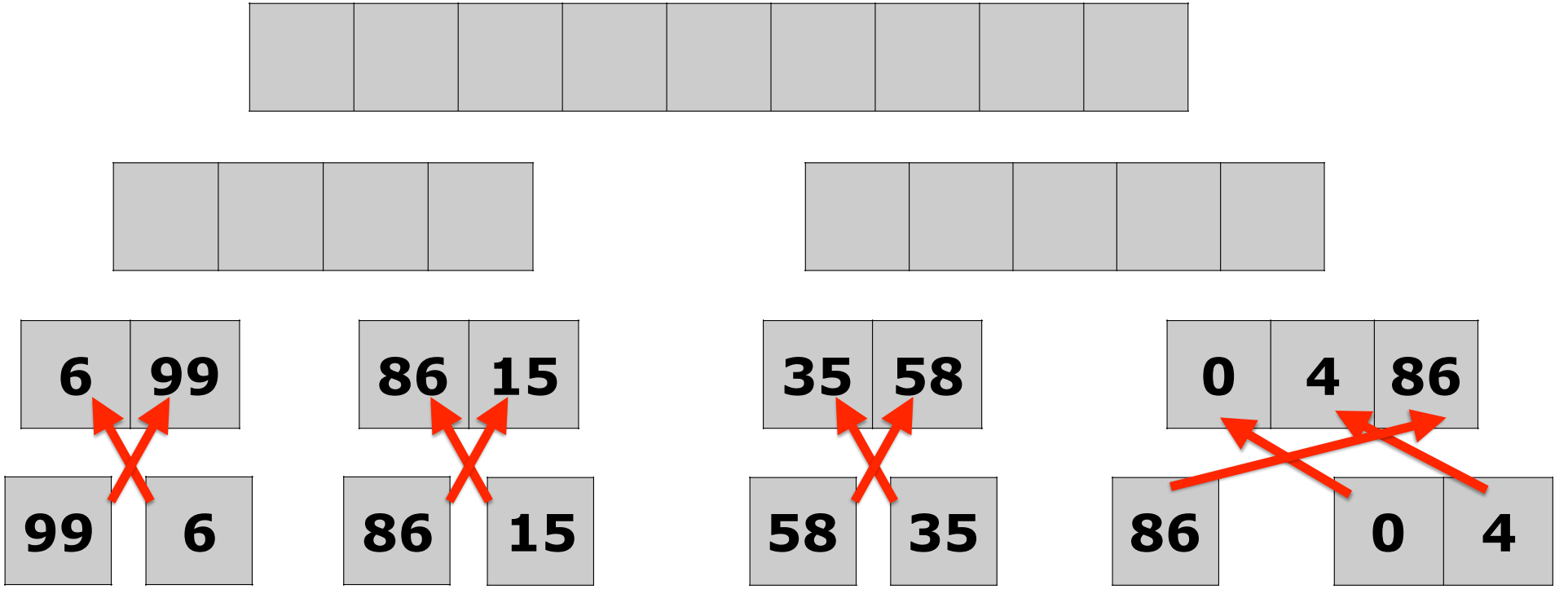


Merge Sort: Full Example



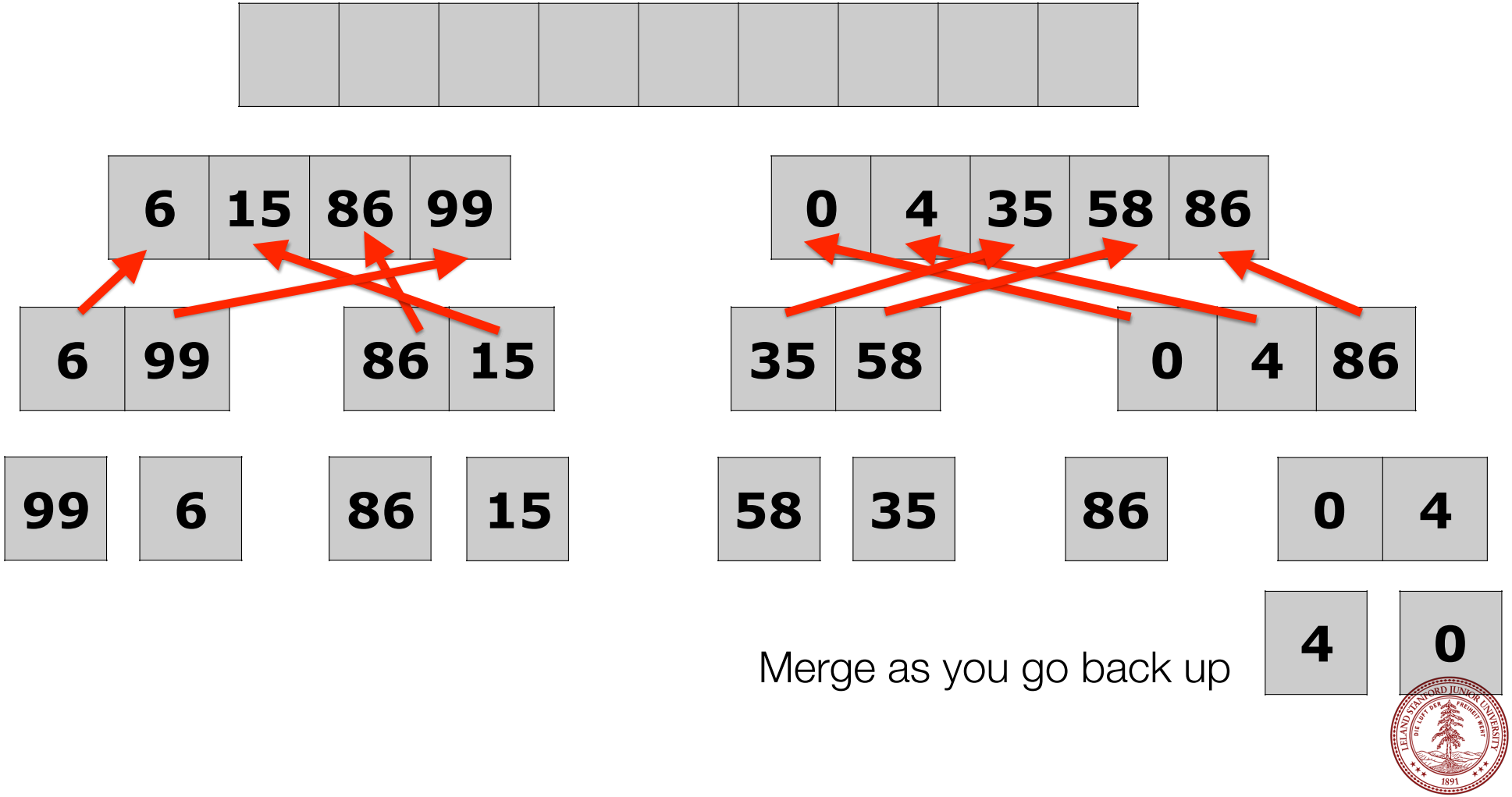
Merge as you go back up

Merge Sort: Full Example

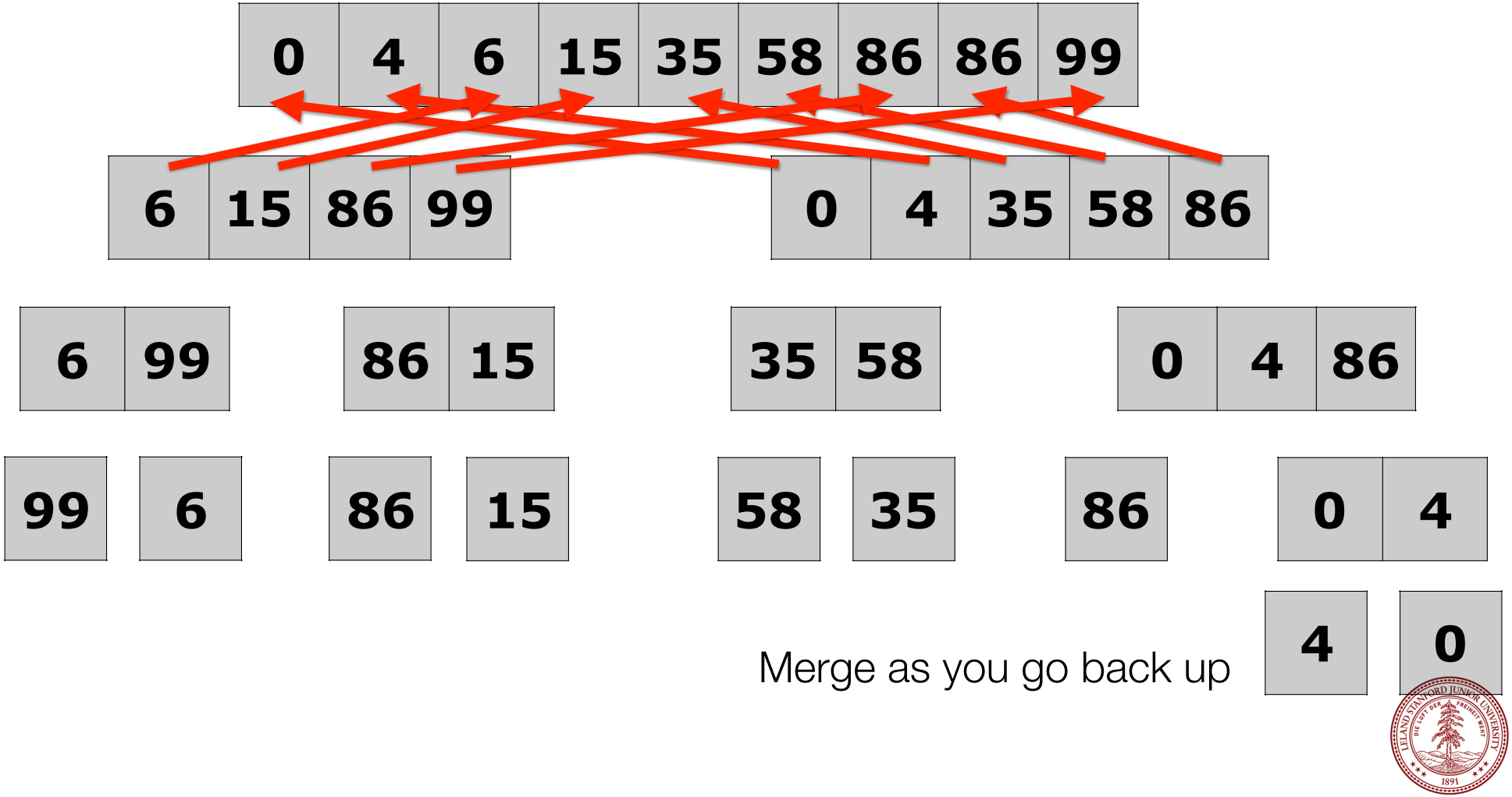


Merge as you go back up

Merge Sort: Full Example



Merge Sort: Full Example



Merge as you go back up

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.

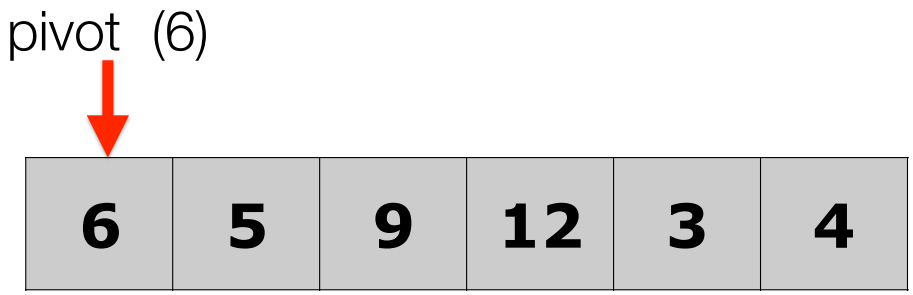
Quicksort Algorithm

- **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.

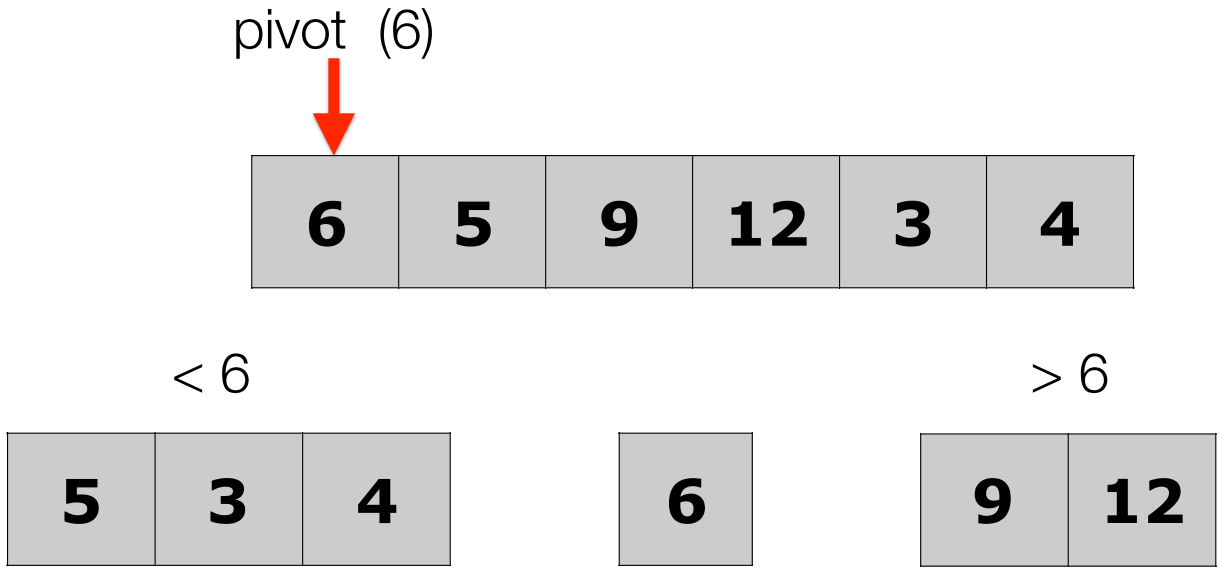
Quicksort Algorithm

- 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

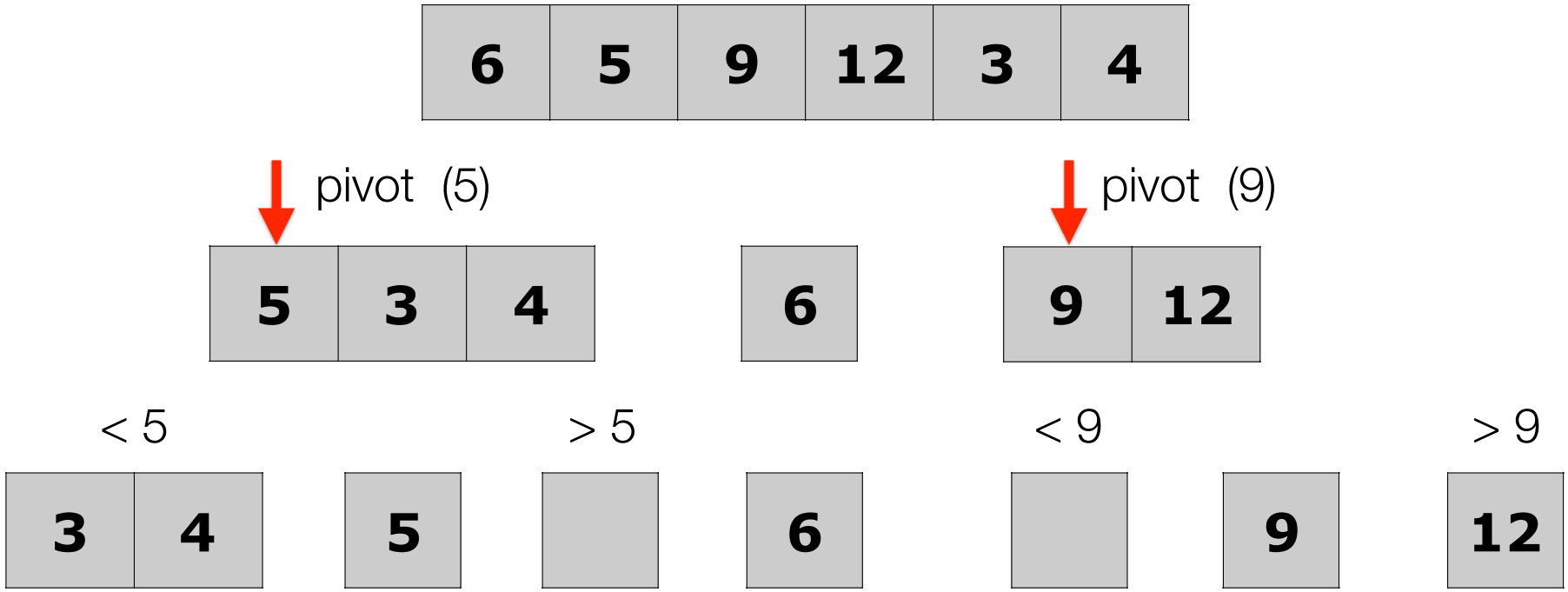


Quicksort Algorithm: Naive



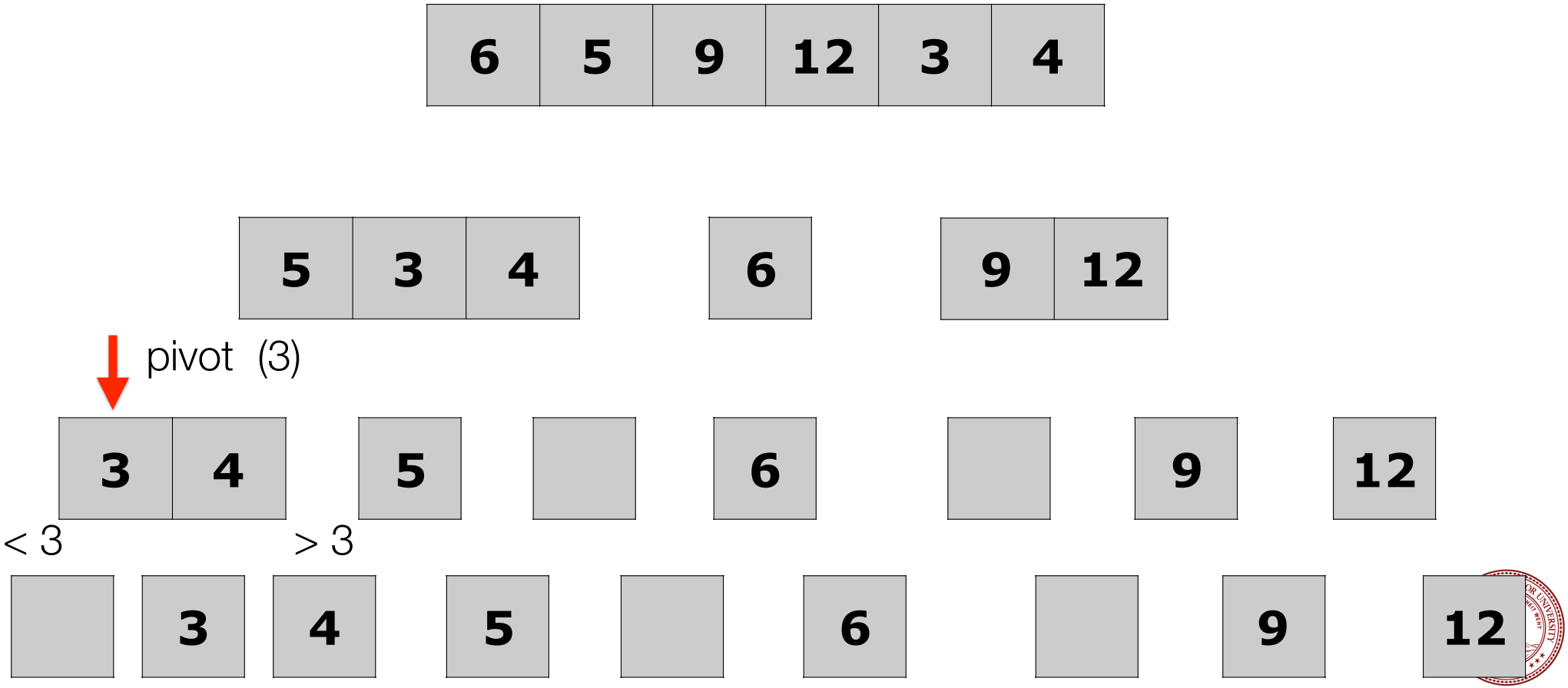
Partition into two new lists -- less than the pivot on the left, and greater than the pivot on the right.
Even if all elements go into one list, that was just a poor partition.

Quicksort Algorithm: Naive

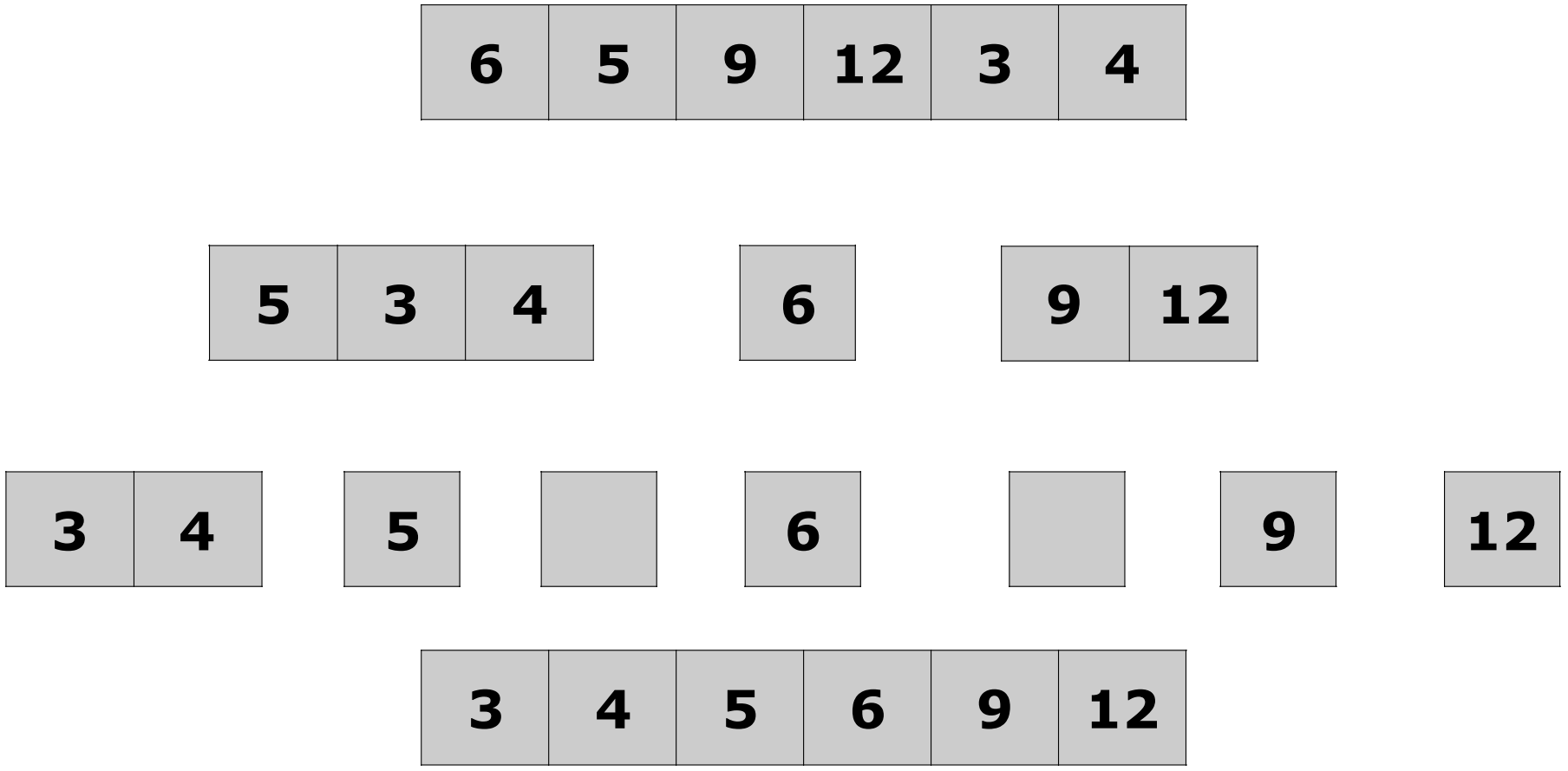


Keep partitioning the sub-lists

Quicksort Algorithm: Naive



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)

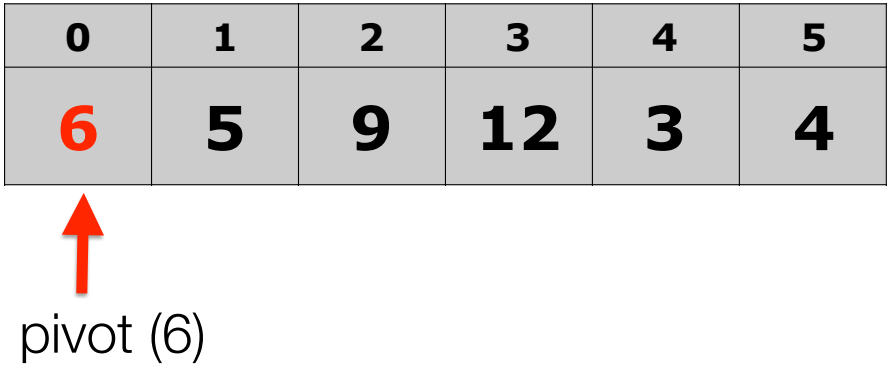
In-place, recursive algorithm:

```
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...

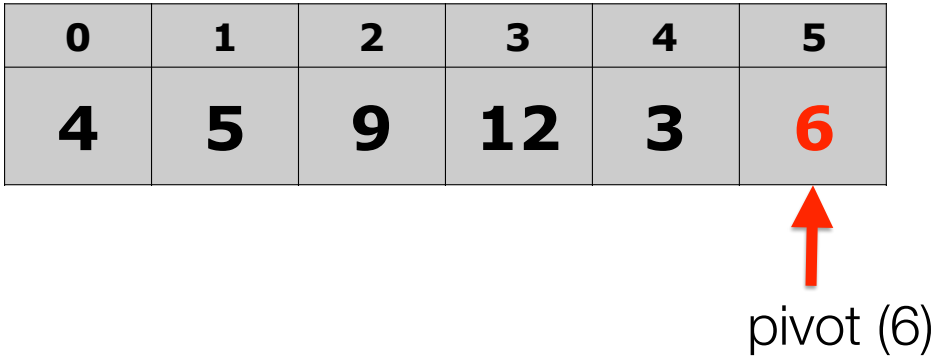
Quicksort Algorithm: In-Place



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

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

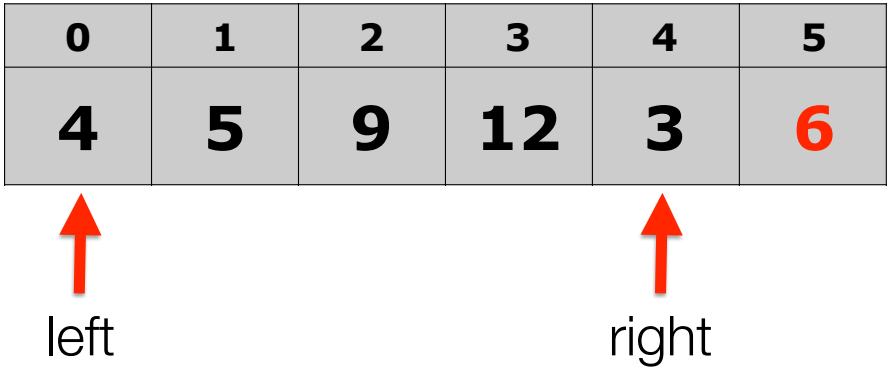
Quicksort Algorithm: In-Place



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

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

Quicksort Algorithm: In-Place

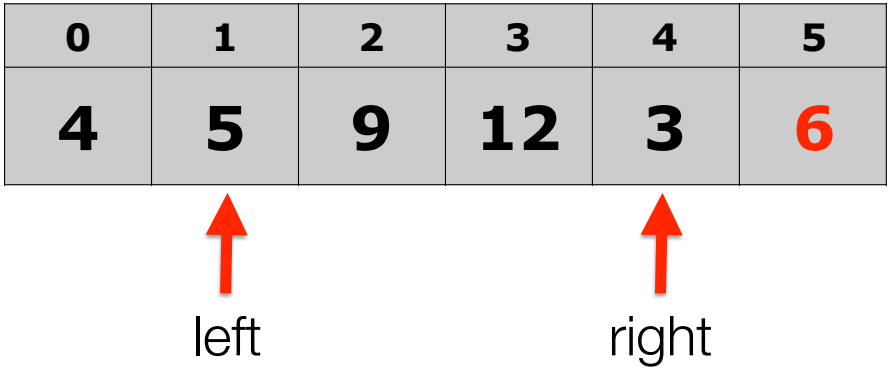


Choose the "left" / "right" indices to be at the start (after the pivot) / end of your vector.

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

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

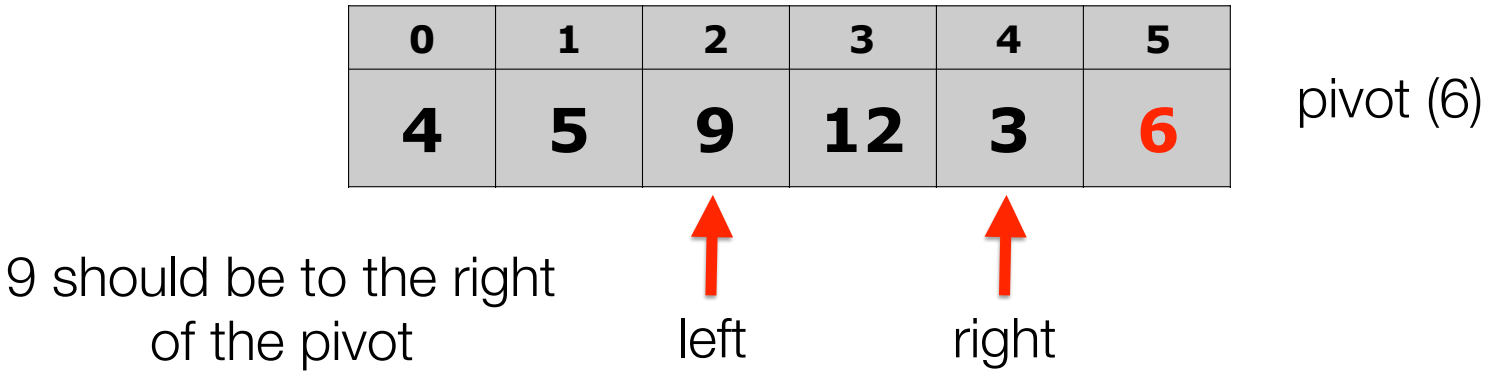
Quicksort Algorithm: In-Place



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

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

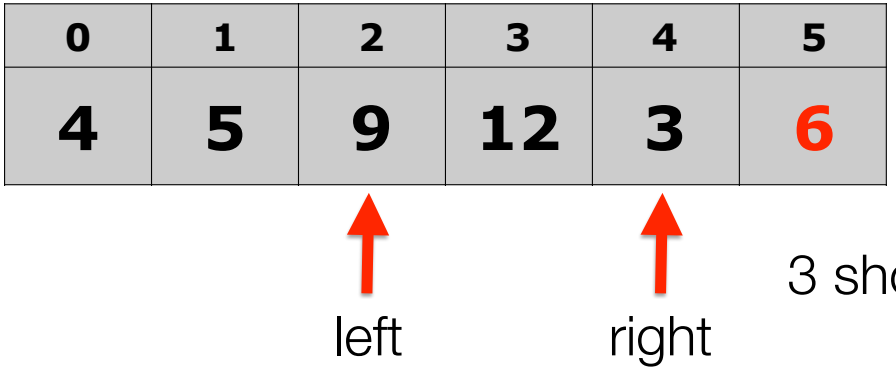
Quicksort Algorithm: In-Place



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

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

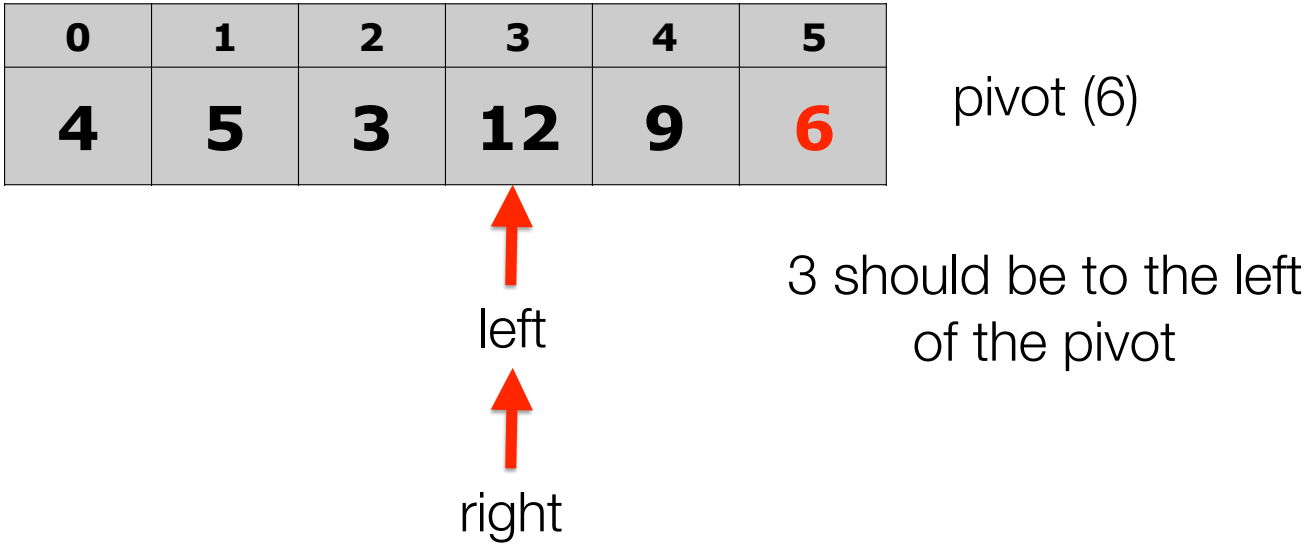
Quicksort Algorithm: In-Place



- 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)
```

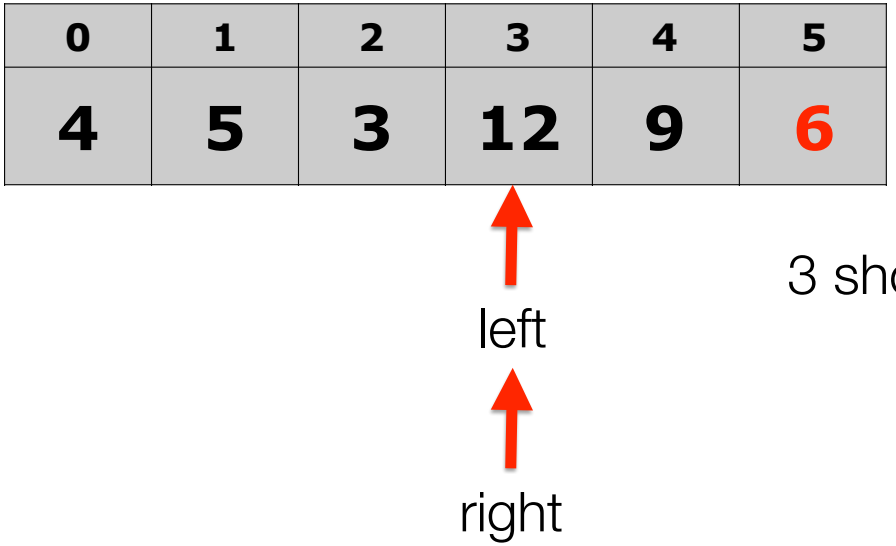
Quicksort Algorithm: In-Place



- 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)
```

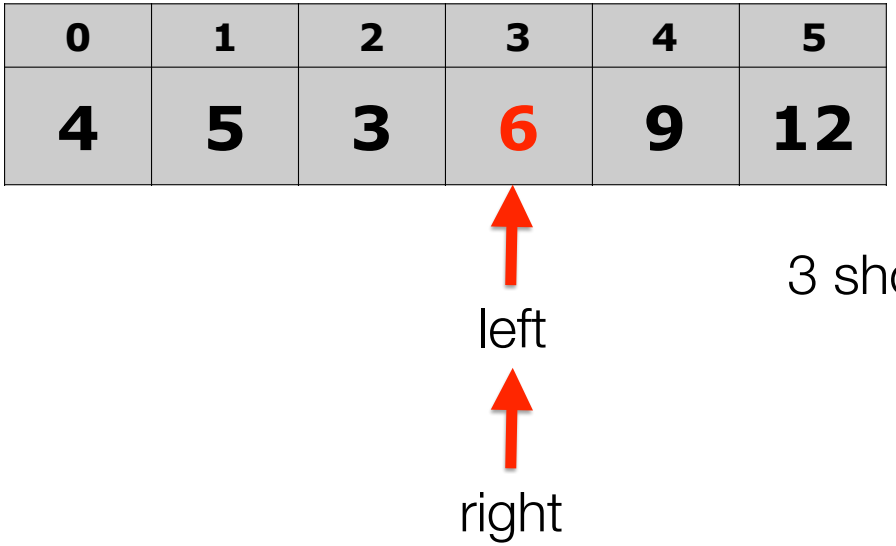
Quicksort Algorithm: In-Place



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



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(n2)
- 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

Radix Sort

- Radix sort is *not* a comparison sort, and works in a completely different manner than other sorts.
- Radix sort uses "buckets" to sort elements, based on which decimal place it is currently working on.
- On each pass, it looks at a different decimal place (ones, tens, hundreds, etc.)
- The "buckets" are individual arrays, and for decimal numbers, we have ten buckets (for the decimal digits 0-9)

Radix Sort: Example (Least Significant Bit first)

Pass 1: units digit

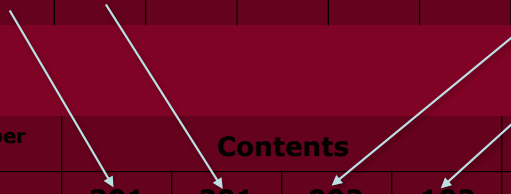
310	213	023	130	013	301	222	032	201	111	323	002	330	102	231	120

Bucket Number	Contents				Units Digit
0	310	130	330	120	0
1	301	201	111	231	1
2	222	032	002	102	2
3	213	023	013	3232	3

Radix Sort: Example (Least Significant Bit first)

Pass 2: tens digit (after concatenating buckets)

310	130	330	120	301	201	111	231	222	032	002	102	213	023	013	323



Bucket Number	Contents				Tens Digit
0	301	201	002	102	0
1	310	111	213	013	1
2	120	222	023	323	2
3	130	330	231	032	3

Radix Sort: Example (Least Significant Bit first)

Pass 3: hundreds digit (after concatenating buckets)

301	201	002	102	310	111	213	013	120	222	023	323	130	330	231	032

Bucket Number	Contents				Hundreds Digit
0	002	013	023	032	0
1	102	111	120	130	1
2	201	213	222	231	2
3	301	310	323	330	3

Now the values are in order (and need to be concatenated)

Radix Sort

- In practice, the bins are generally queues (see animation Java Applet at https://www.cs.auckland.ac.nz/software/AlgAnim/radixsort.html)
- Time complexity is tricky, because radix sort isn't a comparison algorithm. In general, worst-case is *O(d*n)* for *n* keys which have *d* digits.
- Space complexity: O(d + N) because of necessary buckets.

Radix Sort

- If a radix sort is by most significant bit first, then the ordering is lexicographic, e.g.,
- b, d, e, bed, cat, bale, car sorts to:
  - b, bale, bed, car, cat, d, e
- 1,2,3,4,5,6,7,8,9,10 sorts to:
  - 1,10,2,3,4,5,6,7,8,9 (as if the keys were padded with spaces to the right, e.g., 1 = 1_, 2 = 2_, etc.)

Are there other O(n) Sorting Algorithms?

- What would it mean to have an O(n) sorting algorithm, that will sort a list by *only* going through the original list a single time?
- How do teachers sort tests alphabetically?

Bucket Sort

- There is a sort called "bucket sort" (with a "bucket size" of 1) that can run in O(n) time. There are, however, caveats.
- The idea:
  - If you know the range of possible values, set up an array long enough to hold one of each value (initialized to 0, or initialized with a linked list).
  - Iterate through the list, and simply update the count for each item in the array.
  - At the end, the new array will have the counts for each value, and you must loop through that array to pull out the non-zero values.

Bucket Sort

- Max value: 15

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

First, create an array that can hold up to 15

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

Bucket Sort

- Max value: 15

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

Next, iterate through the original array, and update counts as you reach a value.

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

Bucket Sort

- Max value: 15

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

Now, read off the counts back into the original array.

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

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

Bucket Sort Caveats

- You must know the range, and be able to create an array large enough to hold one of each in the *entire* range. (However, you could have bucket sizes that are greater than one, which would mean that you then sort individually in each bucket, which would be worse performance).
- Asymptotically: O(size of range) not O(n). If you have a huge range but a small number of elements, this is not a good sort.

Bucket Sort Uses

- A teacher who needs to sort a stack of papers alphabetically by student name will generally use a bucket sort to group names by first letter of last names. Then, an insertion sort can be used on each individual bucket.
- This is relatively fast for humans, and generally pretty fast for computers.