OBJECT ORIENTED ANALYSIS & DESIGN
DATA STRUCTURES & ALGORITHMS

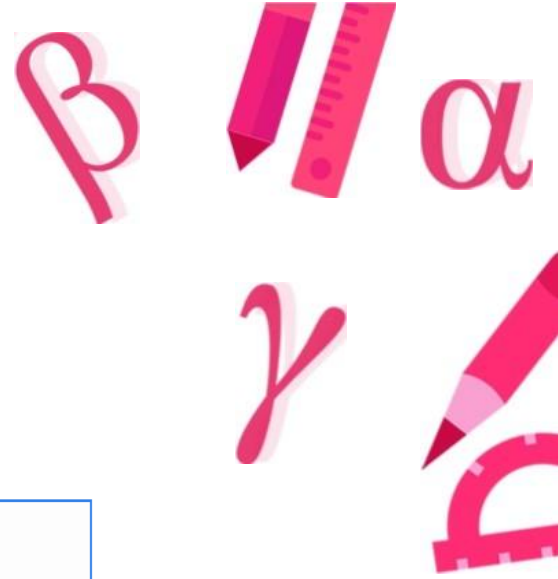
Sorting

Sorting Algorithms

- Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications.
- Your algorithm choice affects only the algorithm's runtime and memory use.
- The next two sections introduce the selection sort and insertion sort—simple algorithms to implement, but not efficient.
- In each case, we examine the efficiency of the algorithms using Big O notation.
- We then present the merge sort algorithm, which is much faster but is more difficult to implement.

Insertion Sort

```
public static void insertionSort(int[] array) {
        for (int i = 1; i < array.length; i++) {
        int current = array[i];
        int j = i - 1;
        while(j >= 0 && current < array[j]) {
                array[j+1] = array[j];
                j--;
        }
        array[j+1] = current; } }
```



```
Unsorted array:
  34   56    4   10   77   51   93   30    5   52
Sorted array:
   4    5   10   30   34   51   52   56   77   93
```

Insertion Sort

Insertion Sort Algorithm

- The algorithm's first iteration takes the array's second element and, if it's less than the first element, swaps it with the first element (i.e., the algorithm inserts the second element in front of the first element).
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
- At the ith iteration of this algorithm, the first i elements in the original array will be sorted.

Insertion Sort

First Iteration

- Lines 33–34 declare and initialize the array named data with the following values:

  34 56 4 10 77 51 93 30 5 52

- Line 42 passes the array to the insertionSort function, which receives the array in parameter items.

- The function first looks at items[0] and items[1], whose values are 34 and 56, respectively.

- These two elements are already in order, so the algorithm continues—if they were out of order, the algorithm would swap them.

Insertion Sort

Second Iteration

- In the second iteration, the algorithm looks at the value of items[2] (that is, 4).
- This value is less than 56, so the algorithm stores 4 in a temporary variable and moves 56 one element to the right.
- The algorithm then determines that 4 is less than 34, so it moves 34 one element to the right.
- At this point, the algorithm has reached the beginning of the array, so it places 4 in items[0].
- The array now is

  4 34 56 10 77 51 93 30 5 52

Insertion Sort

Third Iteration and Beyond

- In the third iteration, the algorithm places the value of items[3] (that is, 10) in the correct location with respect to the first four array elements.
- The algorithm compares 10 to 56 and moves 56 one element to the right because it's larger than 10.
- Next, the algorithm compares 10 to 34, moving 34 right one element.
- When the algorithm compares 10 to 4, it observes that 10 is larger than 4 and places 10 in items[1].
- The array now is

  4 10 34 56 77 51 93 30 5 52
- Using this algorithm, after the ith iteration, the first i + 1 array elements are sorted.
- They may not be in their final locations, however, because the algorithm might encounter smaller values later in the array.

Selection Sort

```
public static void selectionSort(int[] array) {
        for (int i = 0; i < array.length; i++) {
                int min = array[i];
                int minId = i;
                for (int j = i+1; j < array.length; j++) {
                        if (array[j] < min) {
                                min = array[j];
                                minId = j;
                        }
                }
                // swapping
                int temp = array[i];
                array[i] = min;
                array[minId] = temp;
        }
}
```

Selection Sort

Selection Sort Algorithm

- The algorithm's first iteration of the algorithm selects the smallest element value and swaps it with the first element's value.
- The second iteration selects the second-smallest element value (which is the smallest of the remaining elements) and swaps it with the second element's value.
- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element's value, leaving the largest value in the last element.
- After the ith iteration, the smallest i values will be sorted into increasing order in the first array elements.

Selection Sort (cont.)

First Iteration

- Lines 32–33 declare and initialize the array named data with the following values:

  34 56 4 10 77 51 93 30 5 52

- The selection sort first determines the smallest value (4) in the array, which is in element 2.

- The algorithm swaps 4 with the value in element 0 (34), resulting in

  4 56 34 10 77 51 93 30 5 52

Selection Sort (cont.)

Second Iteration

The algorithm then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8.

The program swaps the 5 with the 56 in element 1, resulting in

4 5 34 10 77 51 93 30 56 52

Selection Sort (cont.)

Third Iteration

- On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

```
4   5    10    34    77    51    93    30    56    52
```

- The process continues until the array is fully sorted.

```
4   5    10    30    34    51    52    56    77    93
```

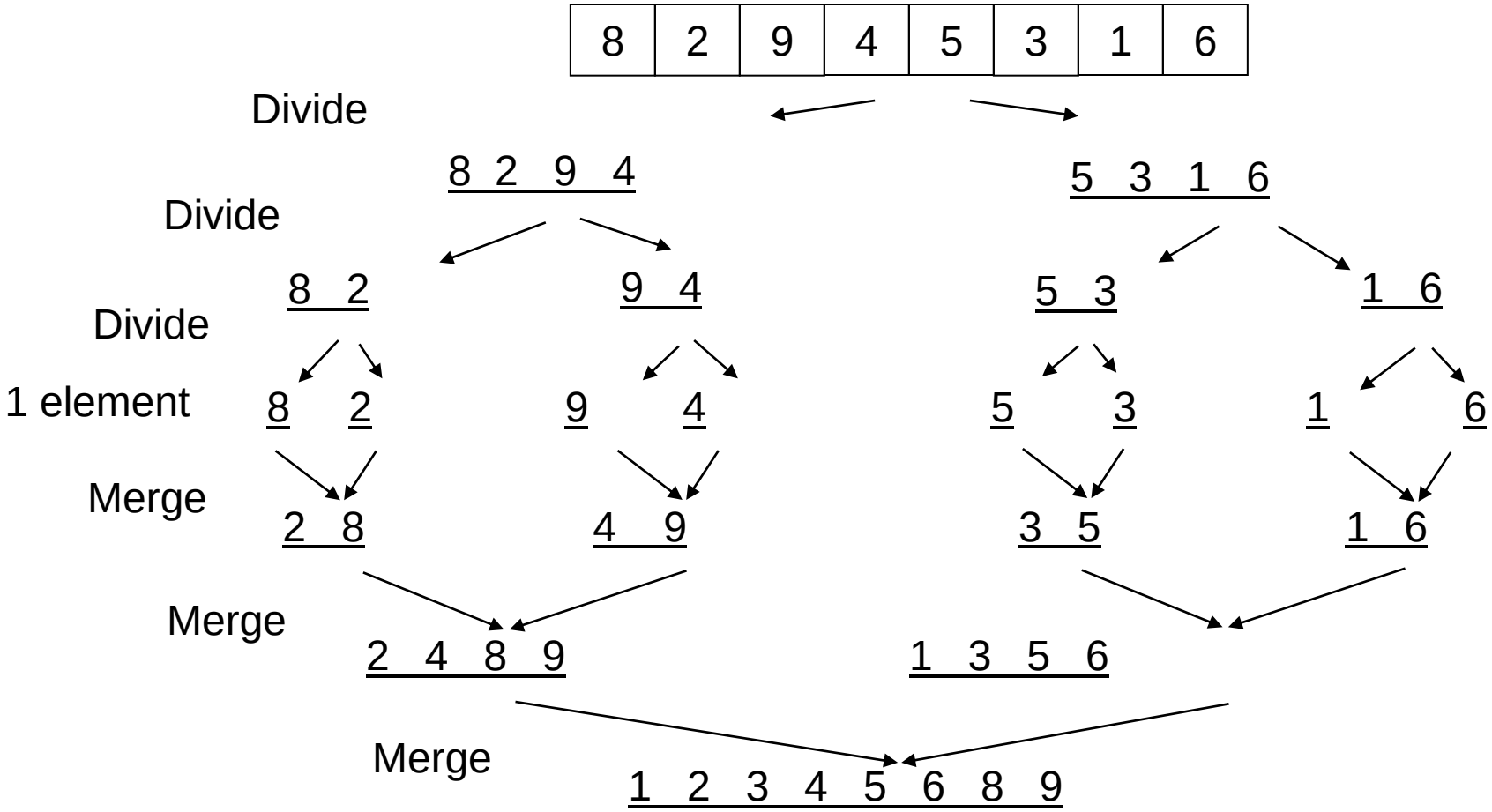
- After the first iteration, the smallest element is in the first position; after the second iteration, the two smallest elements are in order in the first two positions and so on.

Merge Sort

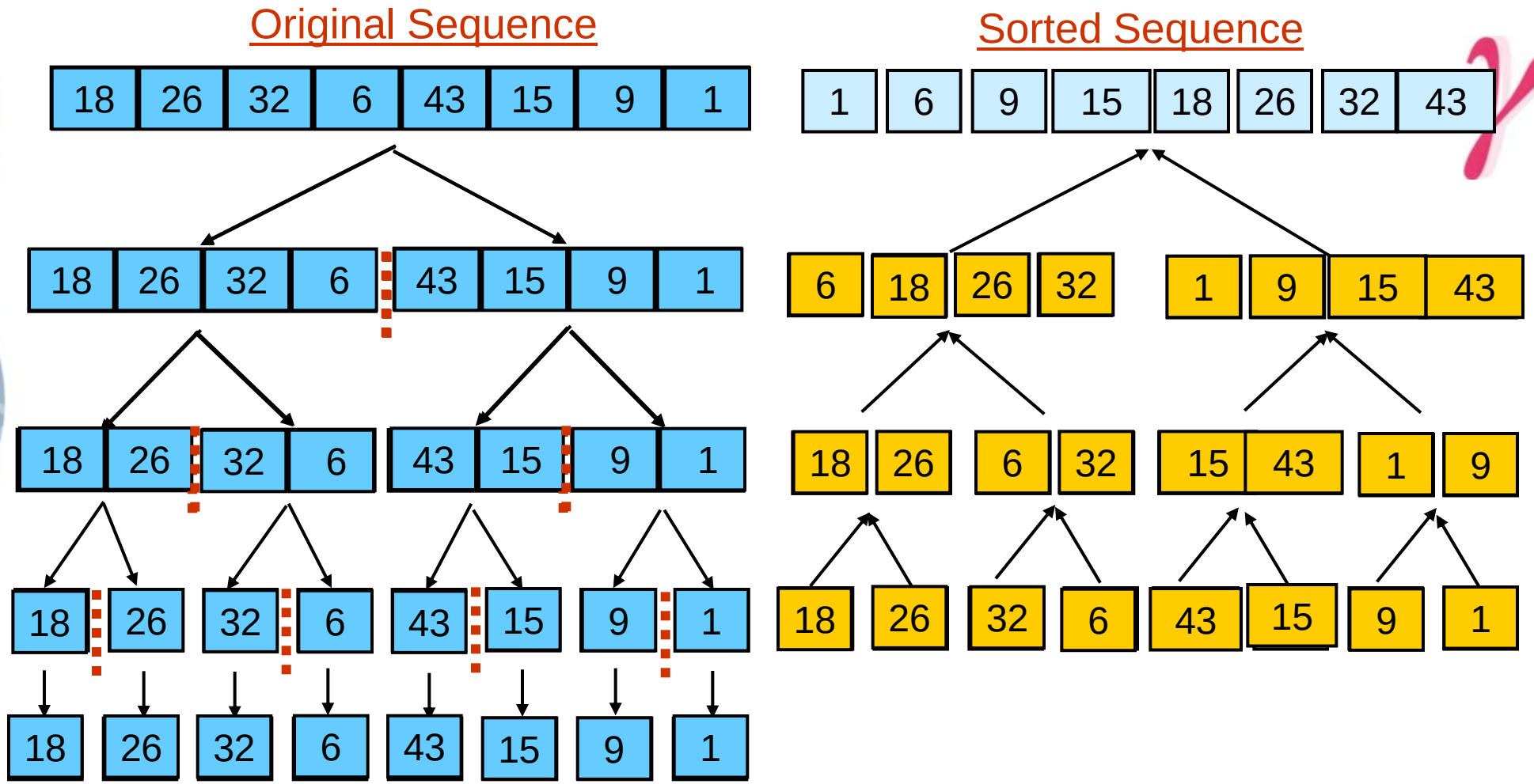
Divide And Conquer

- Merging two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the left side then the right side and then merging the left and right back together.

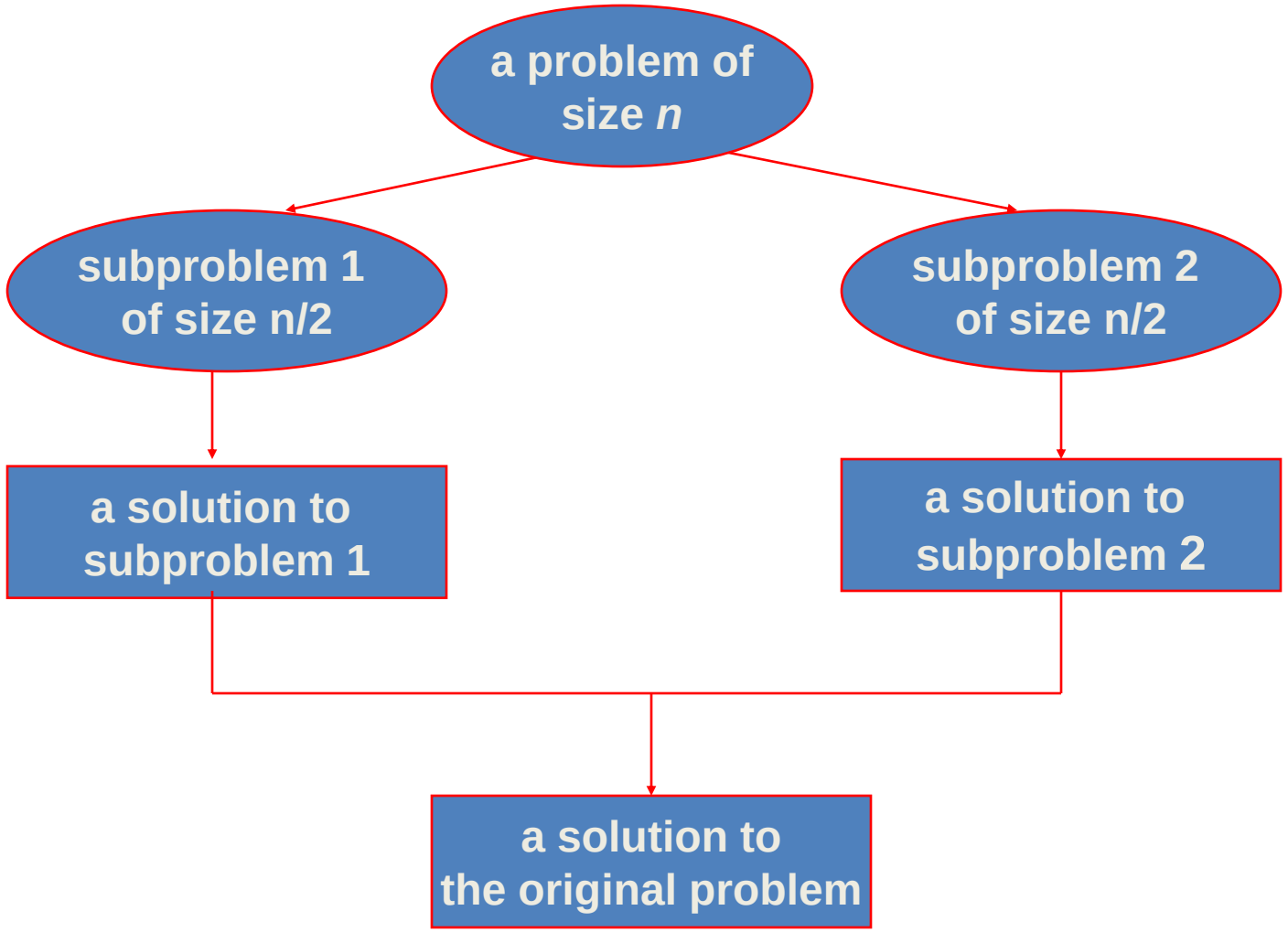
Mergesort Example



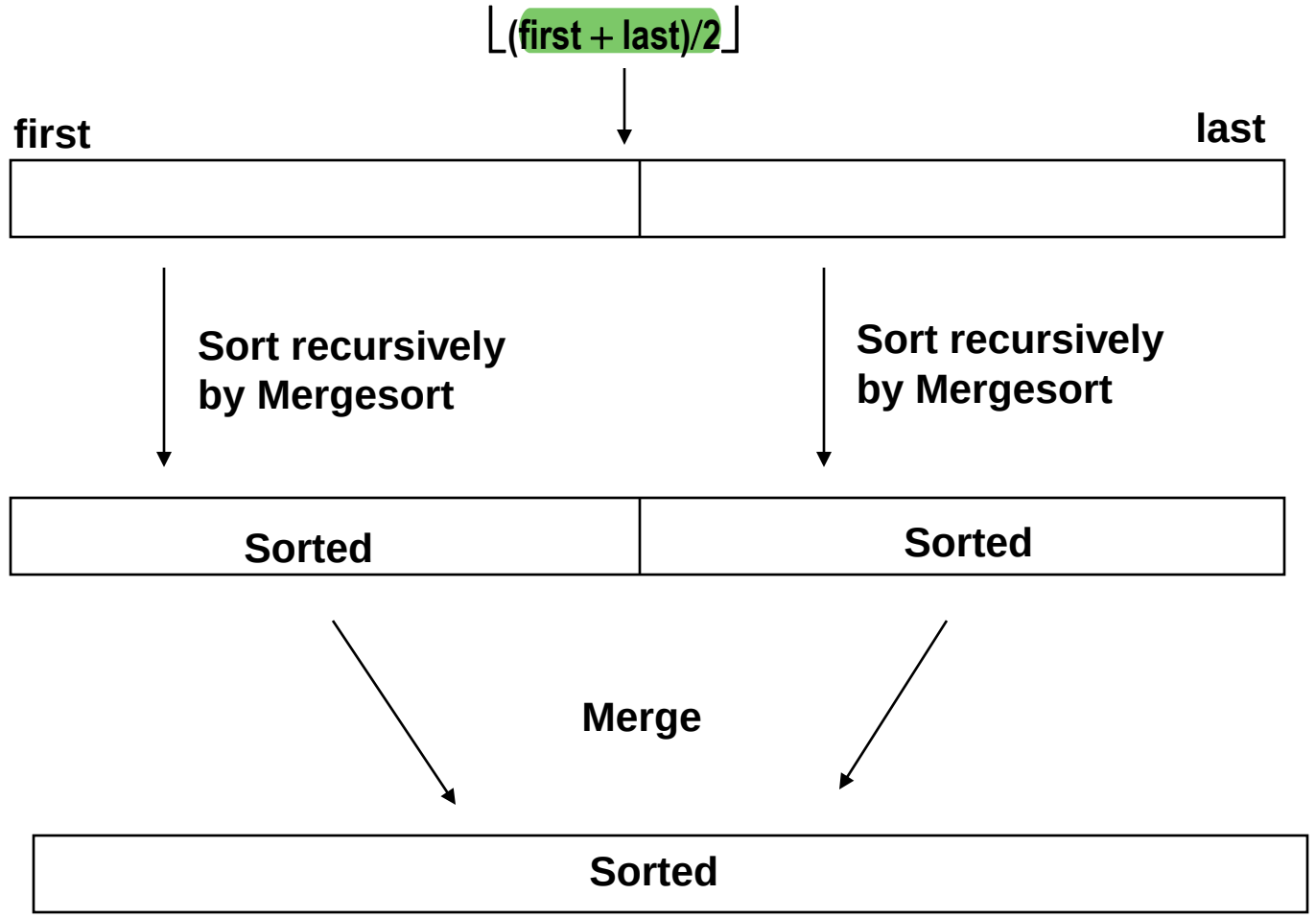
Merge Sort – Example



Divide-and-conquer Technique



Using Divide and Conquer: Mergesort strategy



Merge Sort Algorithm

Divide: Partition ‘n’ elements array into two sub lists with n/2 elements each

Conquer: Sort sub list1 and sub list2

Combine: Merge sub list1 and sub list2

Merge Sort Example

99	6	86	15	58	35	86	4	0

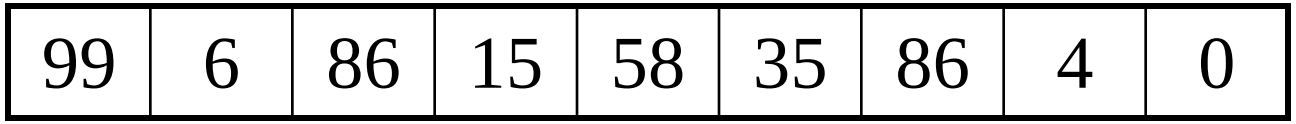
Merge Sort Example

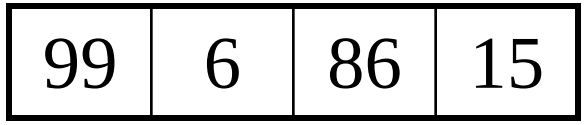
99	6	86	15	58	35	86	4	0

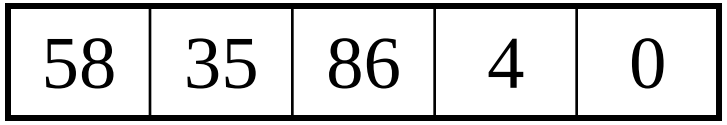
99	6	86	15

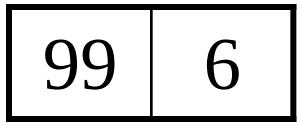
58	35	86	4	0

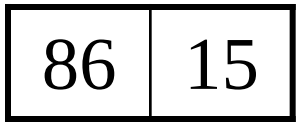
Merge Sort Example

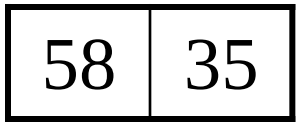


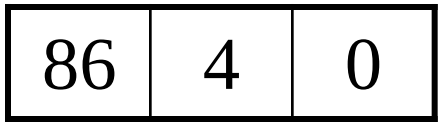




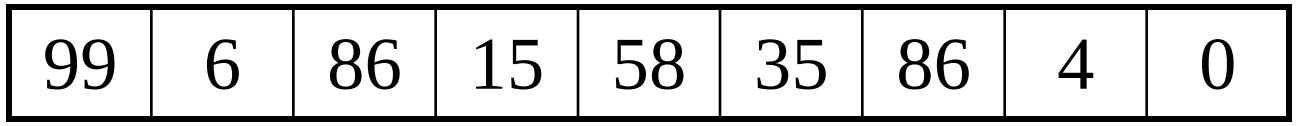


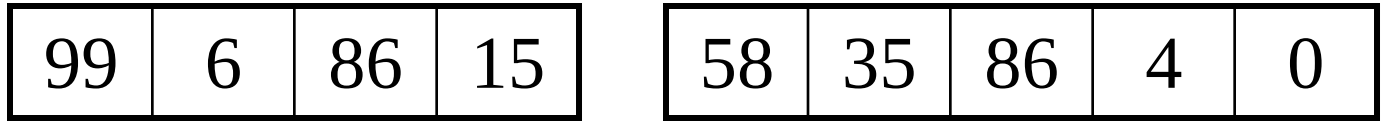


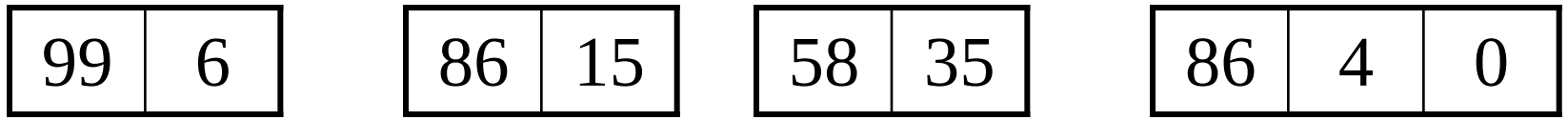


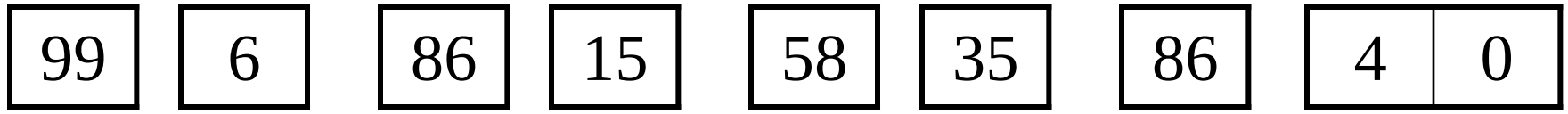


Merge Sort Example

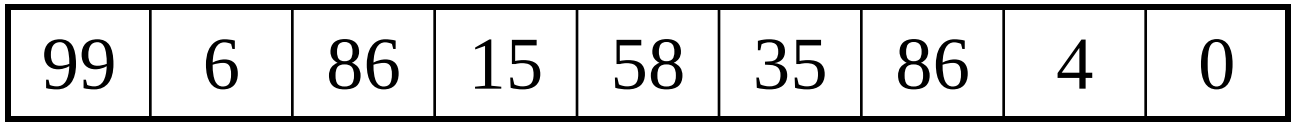


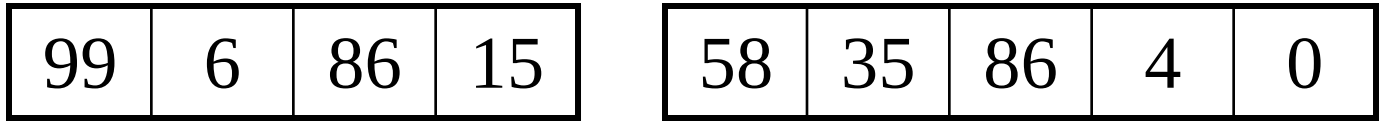


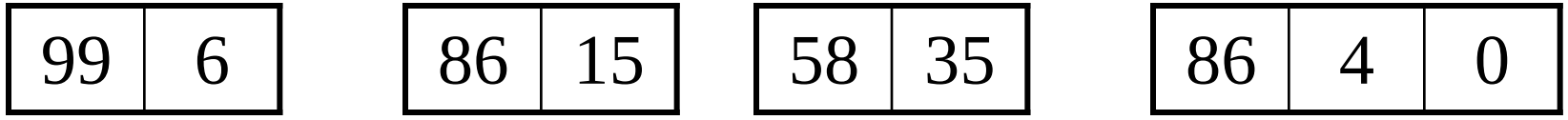


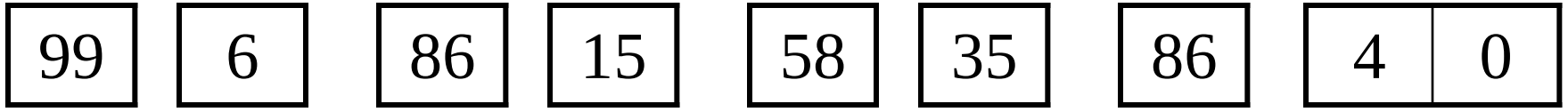


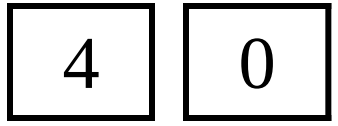
Merge Sort Example



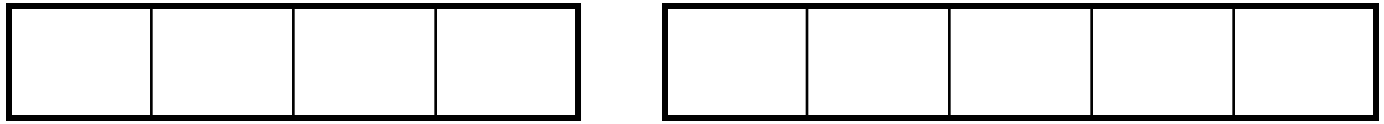


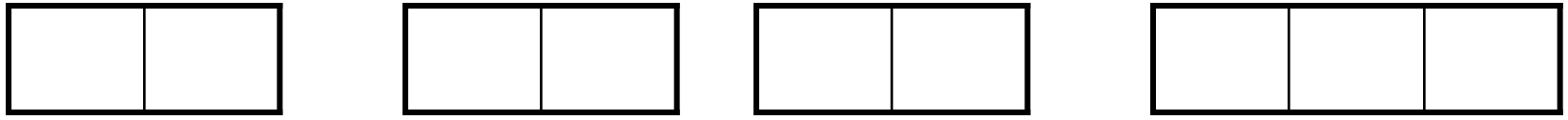


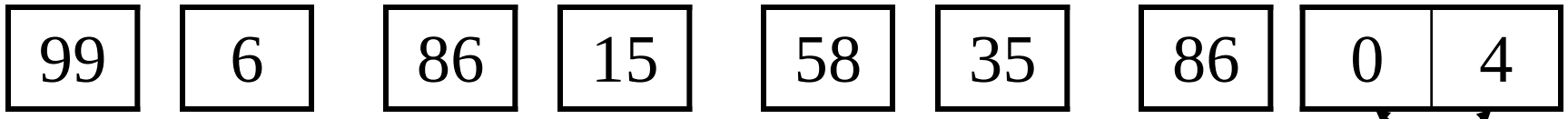




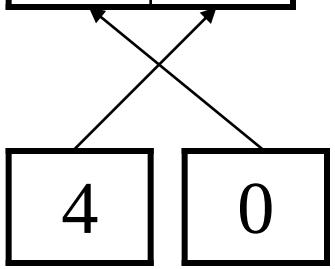
Merge Sort Example



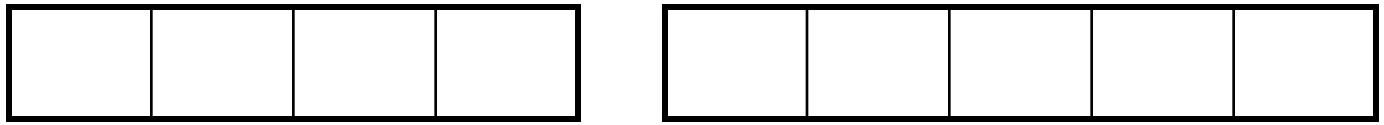


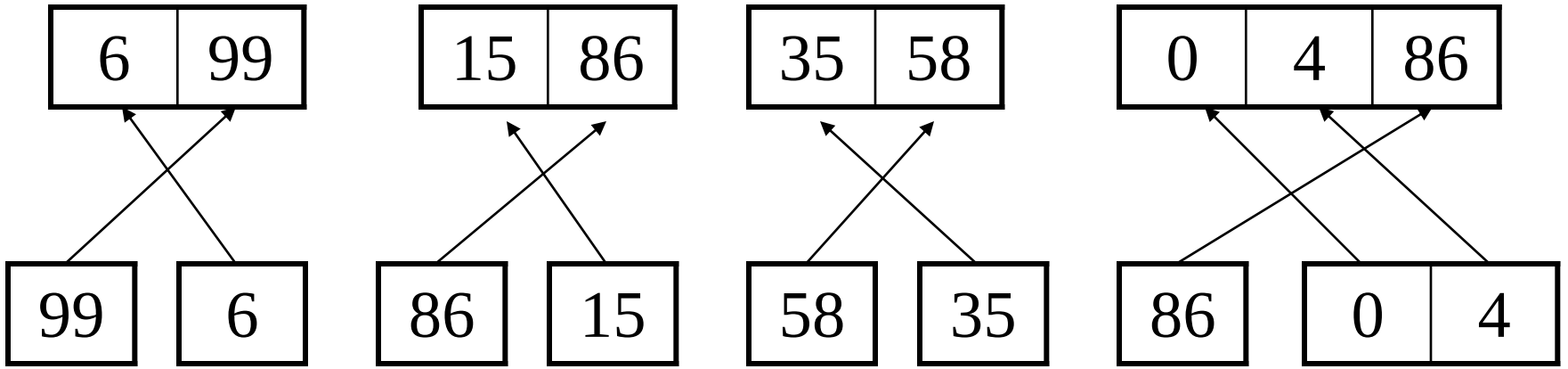


Merge



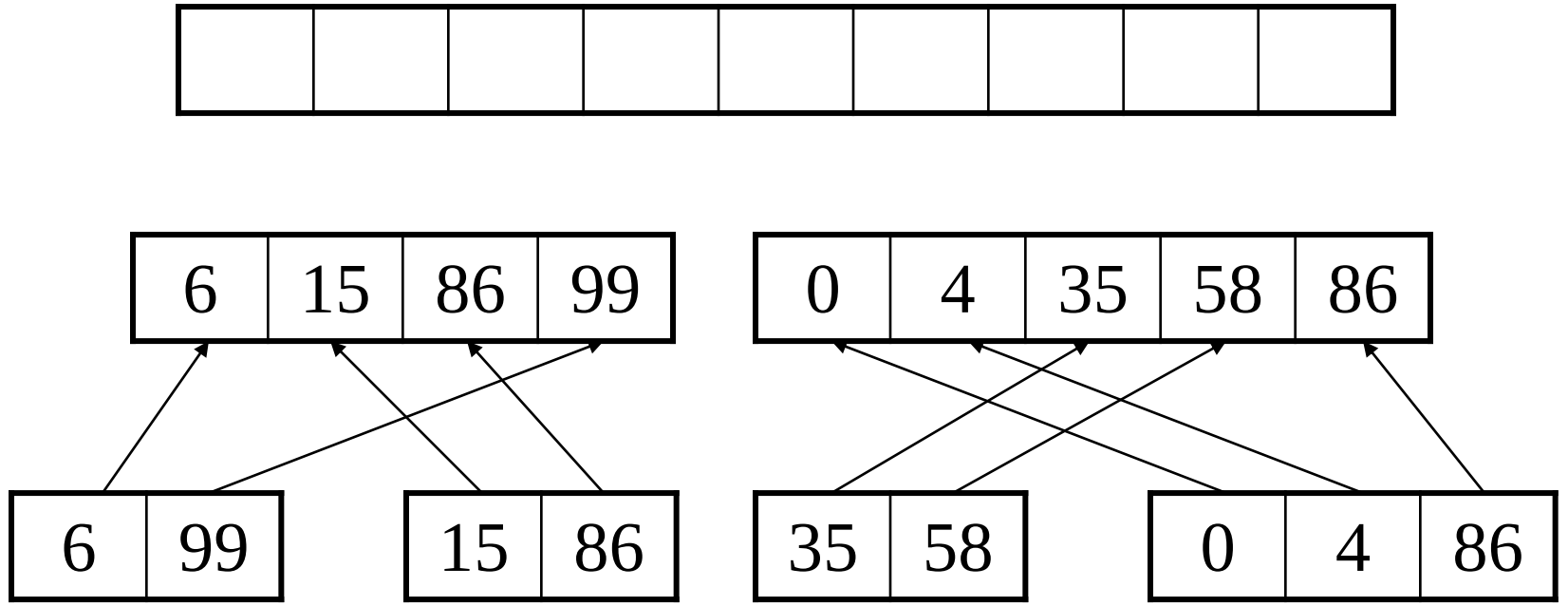
Merge Sort Example





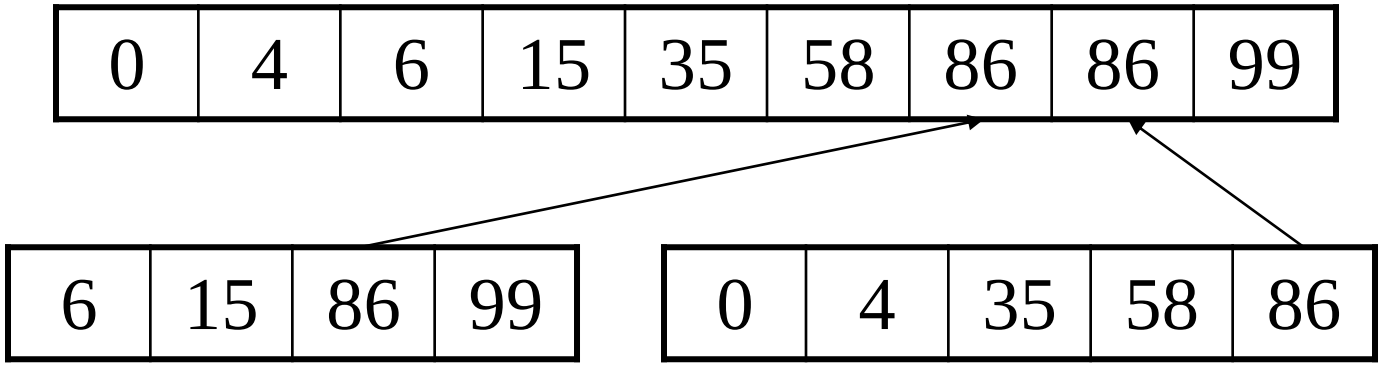
Merge

Merge Sort Example



Merge

Merge Sort Example



Merge

Merge Sort Example

0	4	6	15	35	58	86	86	99

Program

```
public static void mergeSort(int[] array, int left, int right) {
        if (right <= left)
                return;
        int mid = (left+right)/2;
        mergeSort(array, left, mid);
        mergeSort(array, mid+1, right);
        merge(array, left, mid, right);
}
```

Quicksort

Introduction

Fastest known sorting algorithm in practice

Average case: O(N log N) (we don't prove it)

Worst case: $O(N^2)$

But, the worst case seldom happens.

Another divide-and-conquer recursive algorithm, like mergesort

Quicksort

Divide step:

Pick any element (***pivot***) v in S

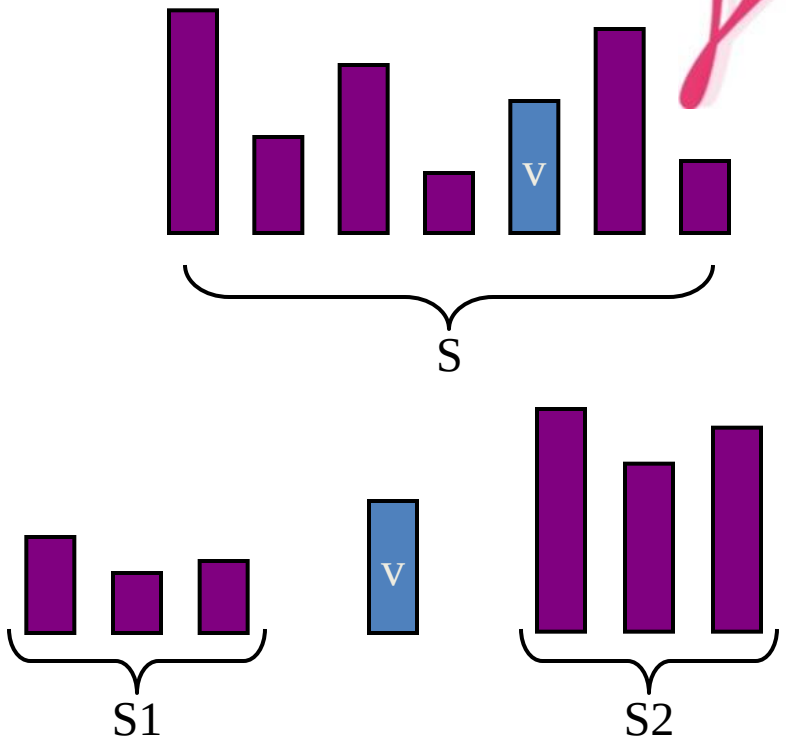
Partition S – {v} into two disjoint groups

S1 = {x ∈ S – {v} | x <= v}

S2 = {x ∈ S – {v} | x ≥ v}

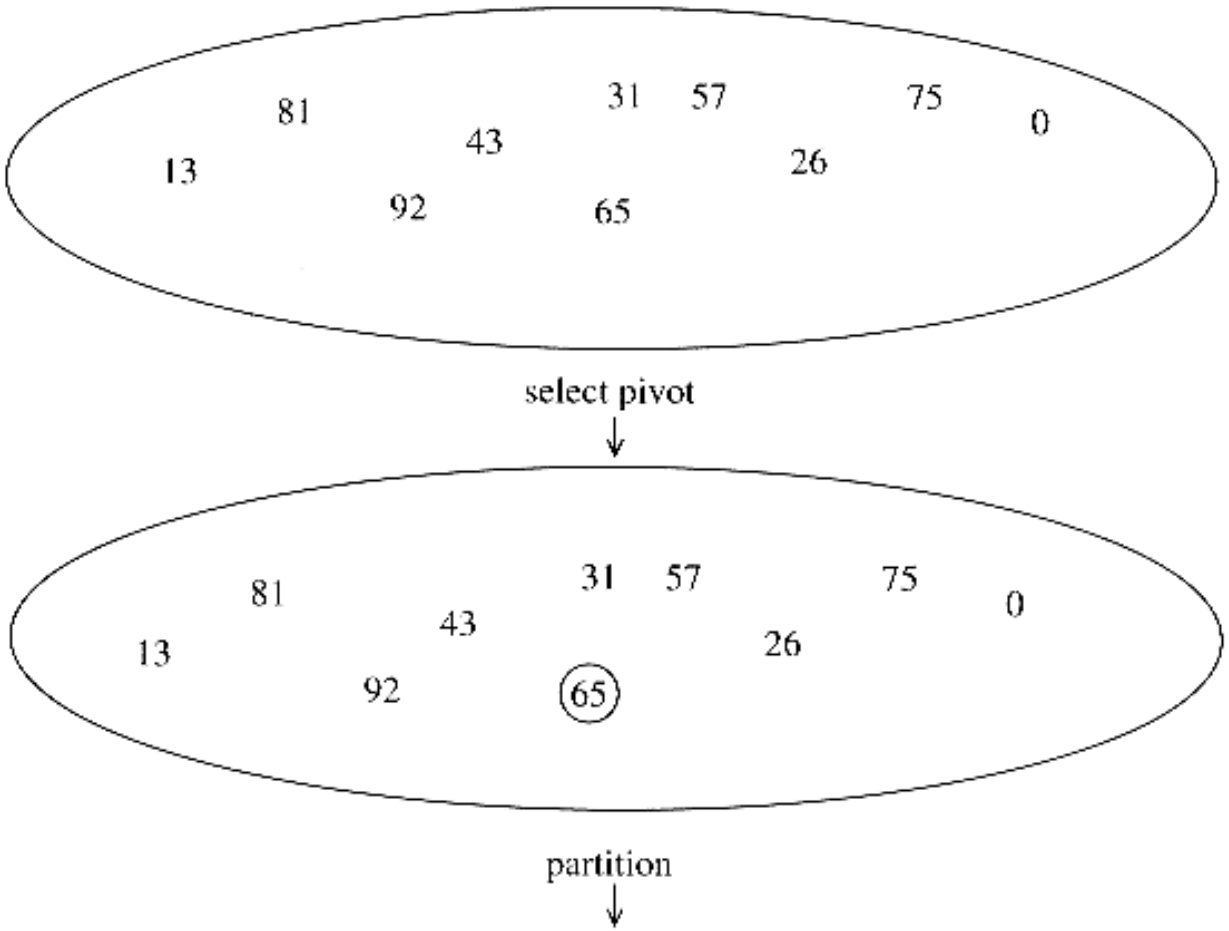
Conquer step: recursively sort S1 and S2

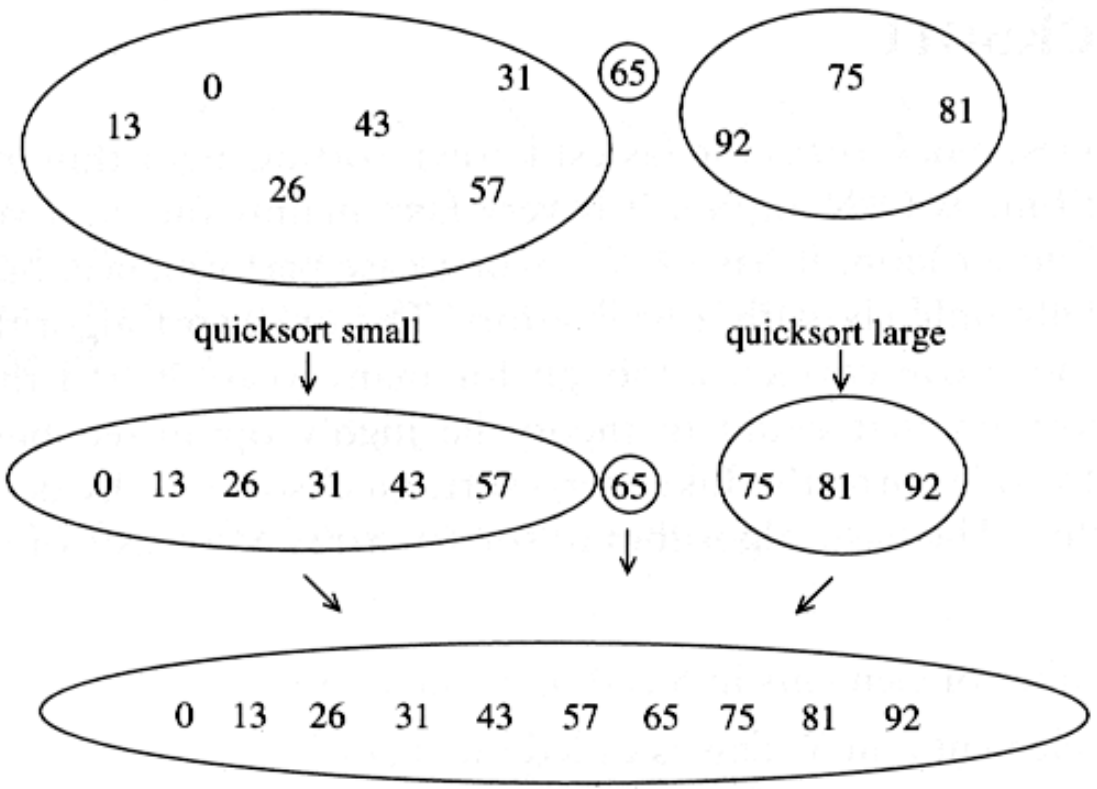
Combine step: the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)



To simplify, we may assume that we don't have repetitive elements,
So to ignore the 'equality' case!

Example


81
31
57
75
0
43
13
26
92
65
select pivot
81
31
57
75
0
43
13
26
92
65
partition


0
31
65
13
43
75
81
92
26
57
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Pseudo-code

```
static int partition(int[] array, int begin, int end) {
        int pivot = end;
        int counter = begin;
        for (int i = begin; i < end; i++) {
                if (array[i] < array[pivot]) {
                        int temp = array[counter];
                        array[counter] = array[i];
                        array[i] = temp;
                        counter++;
                }
        }
        int temp = array[pivot];
        array[pivot] = array[counter];
        array[counter] = temp;
        return counter;
}
public static void quickSort(int[] array, int begin, int end) {
        if (end <= begin) return;
        int pivot = partition(array, begin, end);
        quickSort(array, begin, pivot-1);
        quickSort(array, pivot+1, end);
}
```

Two key steps

- How to pick a pivot?
- How to partition?

Pick a pivot

- Use the first element as pivot
  if the input is random, ok
  if the input is presorted (or in reverse order)
  all the elements go into S2 (or S1)
  this happens consistently throughout the recursive calls
  Results in O(n2) behavior (Analyze this case later)

- Choose the pivot randomly
  generally safe
  random number generation can be expensive

```
quicksort(a,left,right) {

   if (right>left) {
   pivotIndex = left;
   select a pivot value a[pivotIndex];

   pivotNewIndex=partition(a,left,right,pivotIndex);

   quicksort(a,left,pivotNewIndex-1);
   quicksort(a,pivotNewIndex+1,right);
   }
}
```

Small arrays

For very small arrays, quicksort does not perform as well as insertion sort

- how small depends on many factors, such as the time spent making a recursive call, the compiler, etc

Do not use quicksort recursively for small arrays

- Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

A practical implementation

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Thank You!