Object Oriented Programming with Java

Searching & Sorting

Searching

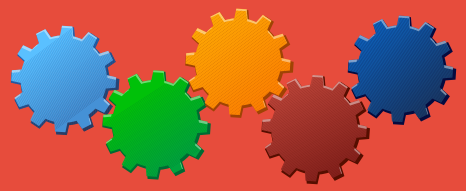
Searching is one of the most common operations in programming

We search a particular item – called target – from a collection of elements

Two main approaches:

- Linear (sequential) search
- Binary search

Practice



There is no doubt you have implemented the linear search way too many times so far.

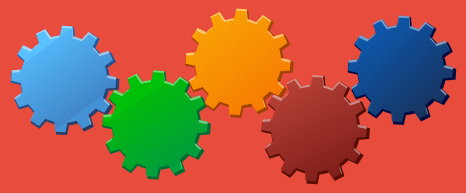
Let's implement it once again but with **generics**

```
public boolean contains(T anEntry)
{
   boolean found = false;

   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(list[index]))
         found = true;
   } // end for

   return found;
} // end contains
```

Practice



There is no doubt you have implemented the linear search way too many times so far.

And now, let's implement it once again with a **recursive** method. You should use generics again.

```
Algorithm to search a[first] through a[last] for desiredItem
if (there are no elements to search)
   return false
else if (desiredItem equals a[first])
   return true
else
   return the result of searching a[first + 1] through a[last]
```

```
private boolean search(int first, int last, T desiredItem)
{
   boolean found;

   if (first > last)
      found = false; // no elements to search
   else if (desiredItem.equals(list[first]))
      found = true;
   else
      found = search(first + 1, last, desiredItem);

   return found;
} // end search
```

Linear Search Efficiency

In an array with ***N*** items, how many steps do we need in the:

Best case scenario? **1 step**

Worst case scenario? **N steps**

Average case? **N/2 steps**

If you had the complete phonebook of the United States, how many steps do you need to find a name by using this linear search approach?

What's a smarter (more efficient) search in this case?

Binary Search

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

Binary Search

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

16 < 18, so search the left half of the array.

Look at the middle entry, 12:

12	15
6	7

16 > 12, so search the right half of the array.

Look at the middle entry, 15:

15
7

16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Linear Search *vs* Binary Search

Binary Search is much faster than Linear Search but it requires **sorted** data.

For a list of *N* numbers, linear search requires ***N* comparisons** in the worst case (*N*/2 on average). Binary search requires **$\log_2(N)$ comparisons** in the worst case.

Examples of scale:

N=1,000:	$\log_2(N) \sim 10$
N=1,000,000:	$\log_2(N) \sim 20$
N=1,000,000,000:	$\log_2(N) \sim 30$

Binary search – Iterative method

How do you search efficiently a sorted list like the phonebook?

```
public int BinarySearch(int[] sortedArray, int key, int low, int high)
{
    while (low <= high)
    {
        int mid = (low + high) / 2;

        if (sortedArray[mid] == key)
            return mid;

        if (sortedArray[mid] < key)
            low = mid + 1;
        else // sortedArray[mid] > key
            high = mid - 1;
    }

    return -1;
}
```

How can we convert it to a generic method?

Binary search – Iterative generic method

```
public <T extends Comparable<T>> int BinarySearch(T[] array, T key, int low, int high)
{
  while (low <= high)
  {
    int mid = (low + high) / 2;

    if (sortedArray[mid].equals(key))
      return mid;

    if (sortedArray[mid].compareTo(key)<0)
      low = mid + 1;
    else // sortedArray[mid] > key
      high = mid - 1;
  }

  return -1;
}
```

How can we convert it to a recursive method?

Binary search – Recursive method

```
public int RecursiveBinarySearch(int[] sortedArray, int key, int low, int high)
{
    int middle = (low + high) / 2;

    if (high < low)
        return -1;

    if (key == sortedArray[middle])
        return middle;
    else if (key < sortedArray[middle])
        return RecursiveBinarySearch(sortedArray, key, low, middle - 1);
    else
        return RecursiveBinarySearch(sortedArray, key, middle + 1, high);
}
```

Built-in methods for binary search

Binary search with **`Arrays`** class

```
int[] arr = {-5, 7, 13, 87};
int key = 4;
java.util.Arrays.binarySearch(arr, key)
```

Binary search with **`Collections`** class

```
java.util.List<Integer> arr = new
java.util.ArrayList<Integer>();
arr.add(-5);
arr.add(7);
arr.add(13);
arr.add(87);
int key = 4;
java.util.Collections.binarySearch(arr, key)
```

Sorting

Sorting

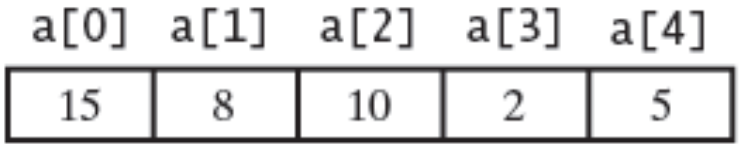
Sorting is one of the most common problems that arises in many applications. Algorithms like *selection sort* and *bubble sort* have a quadratic complexity, i.e. $O(N^2)$, which is often prohibitive for large data sets and real time applications.

Recursive algorithms like *merge sort* and *quicksort* are much faster with an average complexity of $O(N \log_2 N)$

Selection Sort

- Sorting books by height on the shelf
  - Take all books off shelf
  - Select shortest , replace on shelf
  - Continue until all books
- Alternative
  - Look down shelf, select shortest
  - Swap first element with selected shortest
  - Move to second slot, repeat process

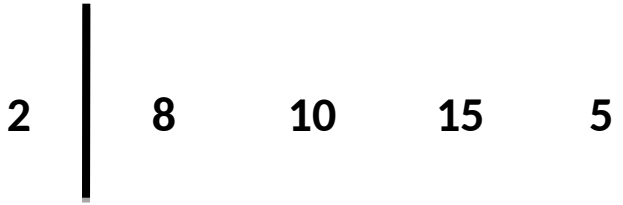
Original array



Pass one

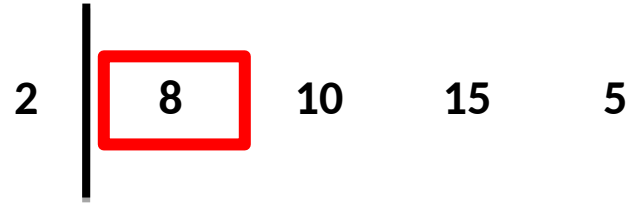
min = 15
minIndex =0

min = 8
minIndex =1



min = 2
minIndex =3

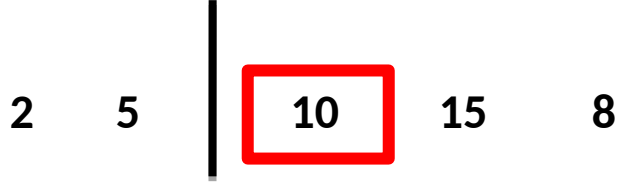
Pass Two



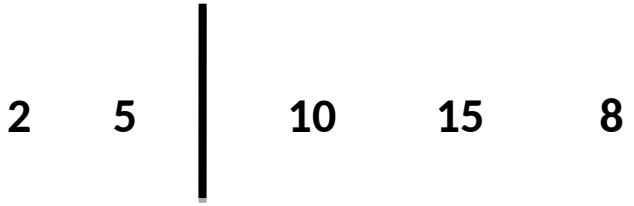
min = 8
minIndex =1

min = 5
minIndex =4

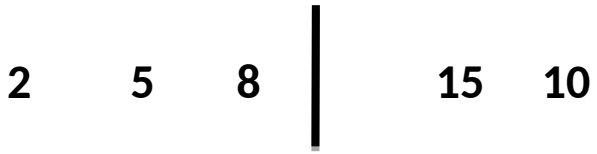
Pass three



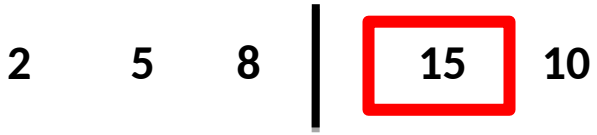
min = 10
minIndex =2



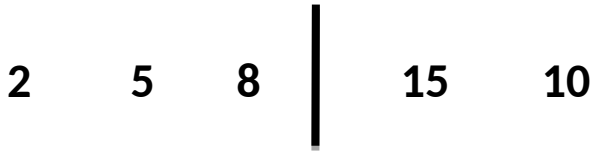
min = 8
minIndex =4



Pass Four



min = 15
minIndex =3



min = 10
minIndex =4

Output

Iterative Selection Sort – Algorithm

Pseudocode for algorithm

```
Algorithm selectionSort(a, n)
// Sorts the first n entries of an array a.

for (index = 0; index < n - 1; index++)
{
   indexOfNextSmallest = the index of the smallest value among
                         a[index], a[index + 1], . . . , a[n - 1]
   Interchange the values of a[index] and a[indexOfNextSmallest]
   // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[index], and these are the smallest
   // of the original array entries. The remaining array entries begin at a[index + 1].
}
```

Iterative Selection Sort – Implementation

```
public static <T extends Comparable<? super T>> void selectionSort(T[] a)
{
   int IndexOfNextSmallest;
   T temp;
   for (int index = 0; index < a.length-1; index++) {
        IndexOfNextSmallest = index;
        T min = a[IndexOfNextSmallest];
        for (int scan = index+1; scan < a.length; scan++){
             if (a[scan].compareTo(min) < 0) {
                  IndexOfNextSmallest = scan;
                  min=a[scan];
             }
        }
        // Swap the values
        temp = a[index];
        a[index]= a[IndexOfNextSmallest];
        a[IndexOfNextSmallest] = temp;
   }
}
```

Recursive Selection Sort – Algorithm

```
Algorithm selectionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
   indexOfNextSmallest = the index of the smallest value among
                         a[first], a[first + 1], . . . , a[last]
   Interchange the values of a[first] and a[indexOfNextSmallest]
   // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[first] and these are the smallest
   // of the original array entries. The remaining array entries begin at a[first + 1].
   selectionSort(a, first + 1, last)
}
```

Recursive Selection Sort – Implementation

```
public static <T extends Comparable<? super T>> void selectionSort ( T[] a, int first, int last){
    if (  first < last ){
         int IndexOfNextSmallest = first;
         T temp;
         T min = a[IndexOfNextSmallest];
         for (int scan = first+1; scan < a.length; scan++){
               if (a[scan].compareTo(min) < 0) {
                     IndexOfNextSmallest = scan;
                     min=a[scan];
               }
         }
         // Swap the values
         temp = a[first];
         a[first]= a[IndexOfNextSmallest];
         a[IndexOfNextSmallest] = temp;
         selectionSort(a, first+1, last);
    }
}
```

Efficiency of Selection Sort

- Efficiency of selection sort is $O(n^2)$ for <u>all cases</u>

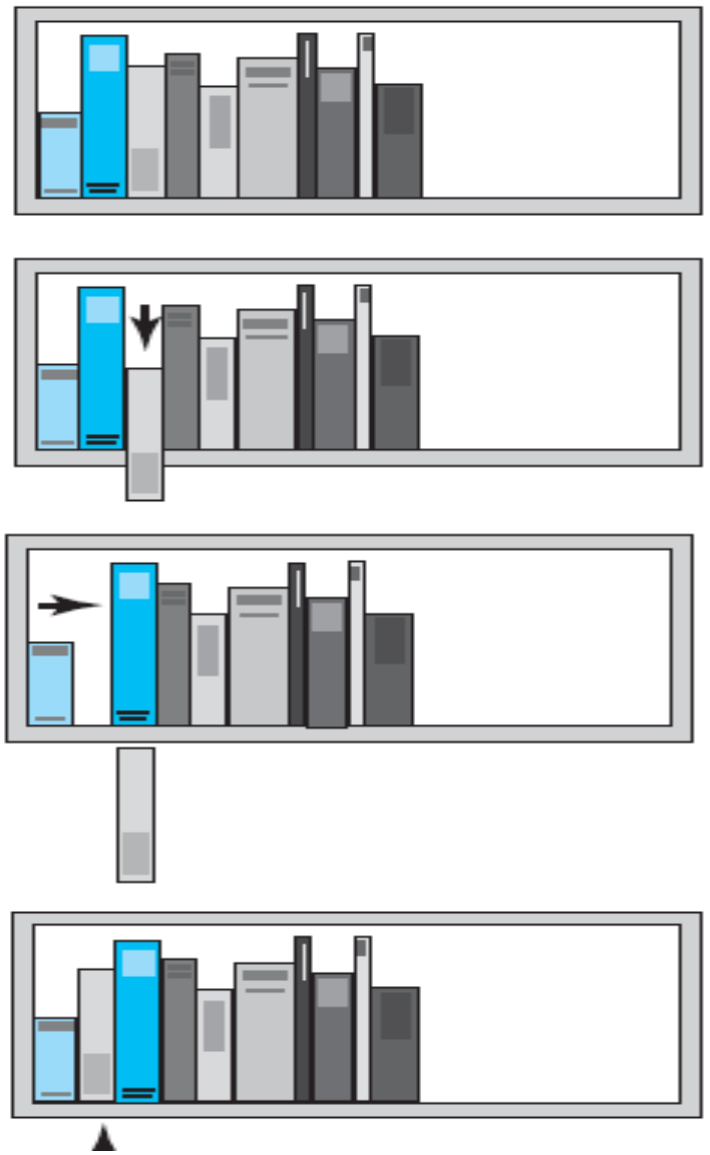
  The inner loop executes:

  $(n-1)+(n-2)+(n-3)+ \ldots\ldots 1 = n(n-1)/2 = (\mathbf{n^2}-n)/2$
- In addition to comparisons, it requires O(n) swaps
- Space complexity is O(1)

Insertion Sort

- When book found taller than one to the right
  - Remove book to right
  - Slide taller book to right
  - Insert shorter book into that spot
- Compare shorter book just moved to left
  - Make exchange if needed
- Continue …

Insertion Sort



Insertion Sort Algorithm

- Partitions the array in to sorted and unsorted
  - Initially the sorted part contains only the first element
  - The unsorted contains all the other elements
- At each pass
  - Removes the first entry from the unsorted part and inserts it into its proper sorted position within the sorted part.

Original Data

a[0]	a[1]	a[2]	a[3]	a[4]
15	8	10	2	5

Pass 1

15 | 8 10 2 5

Pass 2

15 | 8 10 2 5

Pass 3

8 15 | 10 2 5

Pass 4

8 10 15 | 2 5

2 8 10 15 | 5

2 5 8 10 15

Insertion Sort

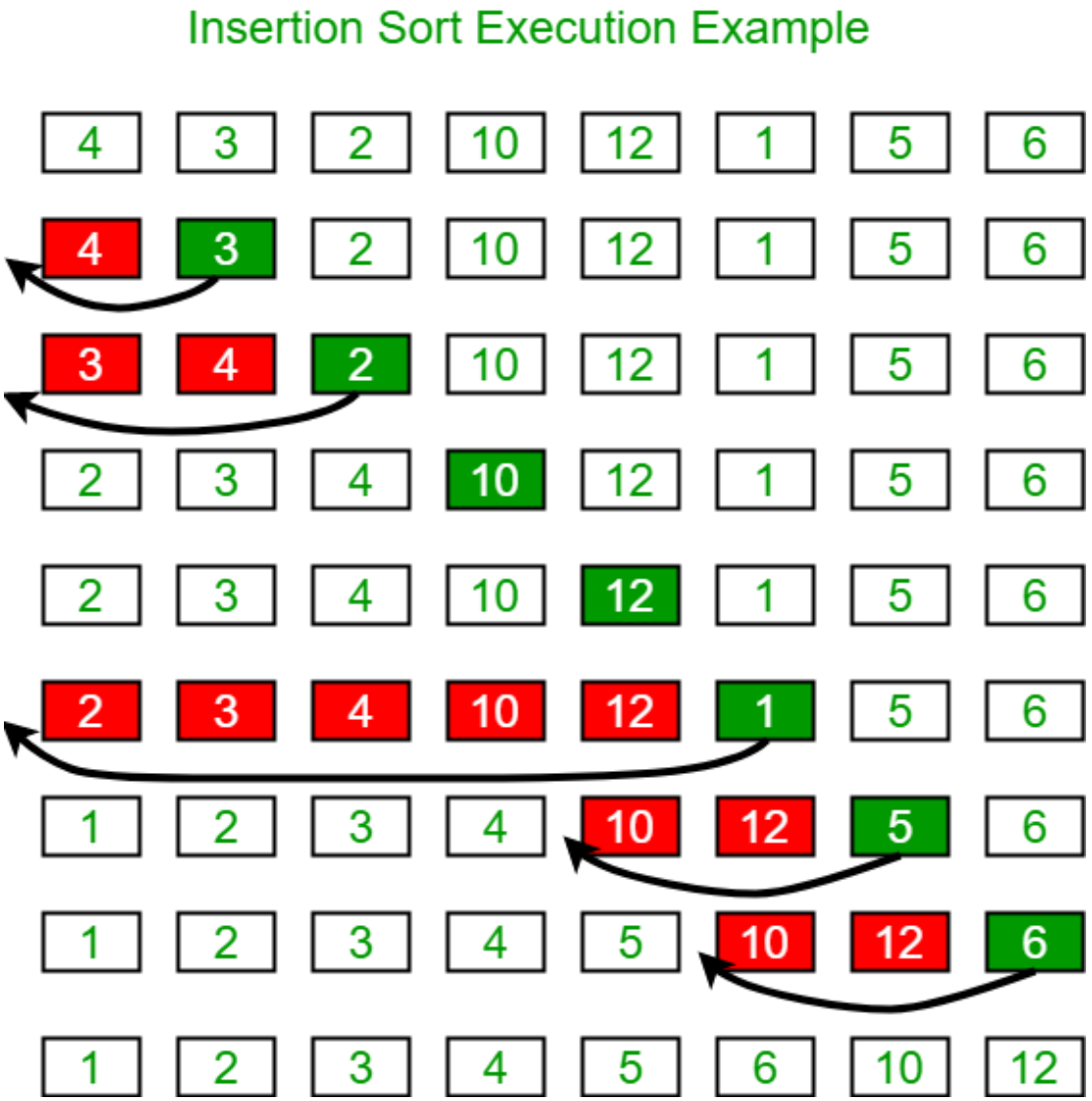


Image source: GeeksforGeeks.org

Insertion Sort

```java
public static <T extends Comparable<? super T>> void insertionSort(T[] a)
{
   for (int index = 1; index < end; index++){
        T min = a[index];
        int position = index;
        // Shift larger values to the right
        while (position > 0 && min.compareTo(a[position-1]) < 0)
        {
             a[position] = a[position-1];
             position--;
        }
        a[position] = min; // put min in its final location
   }
}
```

Efficiency of Insertion Sort

- Efficiency
  - Loop executes at most
    1 + 2 + … (n – 1) times

    Sum is $\frac{n \cdot (n-1)}{2}$
  - Which gives $O(n^2)$
- Best case – array already in order, O(n)

Faster Sorting Algorithms

- **Insertion sort** and **selection sort** are ok when the array size is small
- But when we need to sort large arrays, those methods take much time
- We need better sorting algorithms

Comparison of Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	O(n^2)	O(log(n))
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$

Merge Sort

MergeSort is a "divide and conquer" algorithm

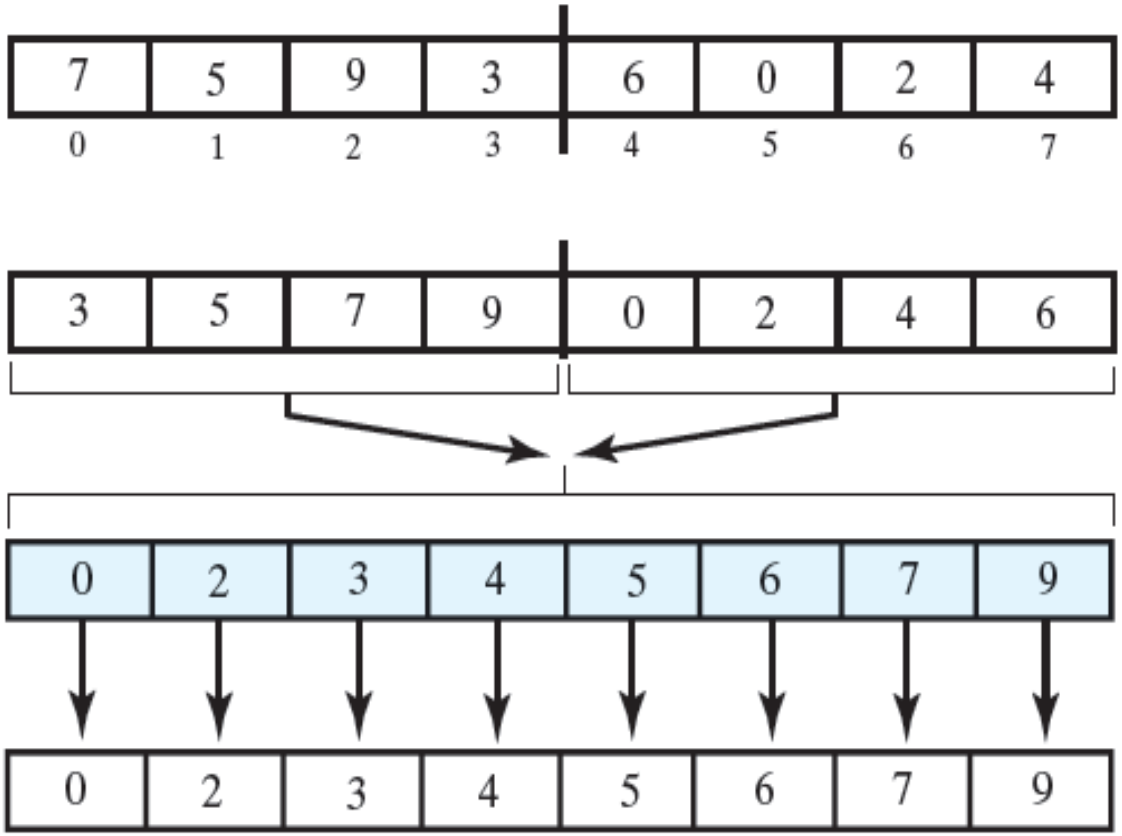
It successively splits the list in half, until each sublist is size 1

Then the two halves are MergeSorted individually

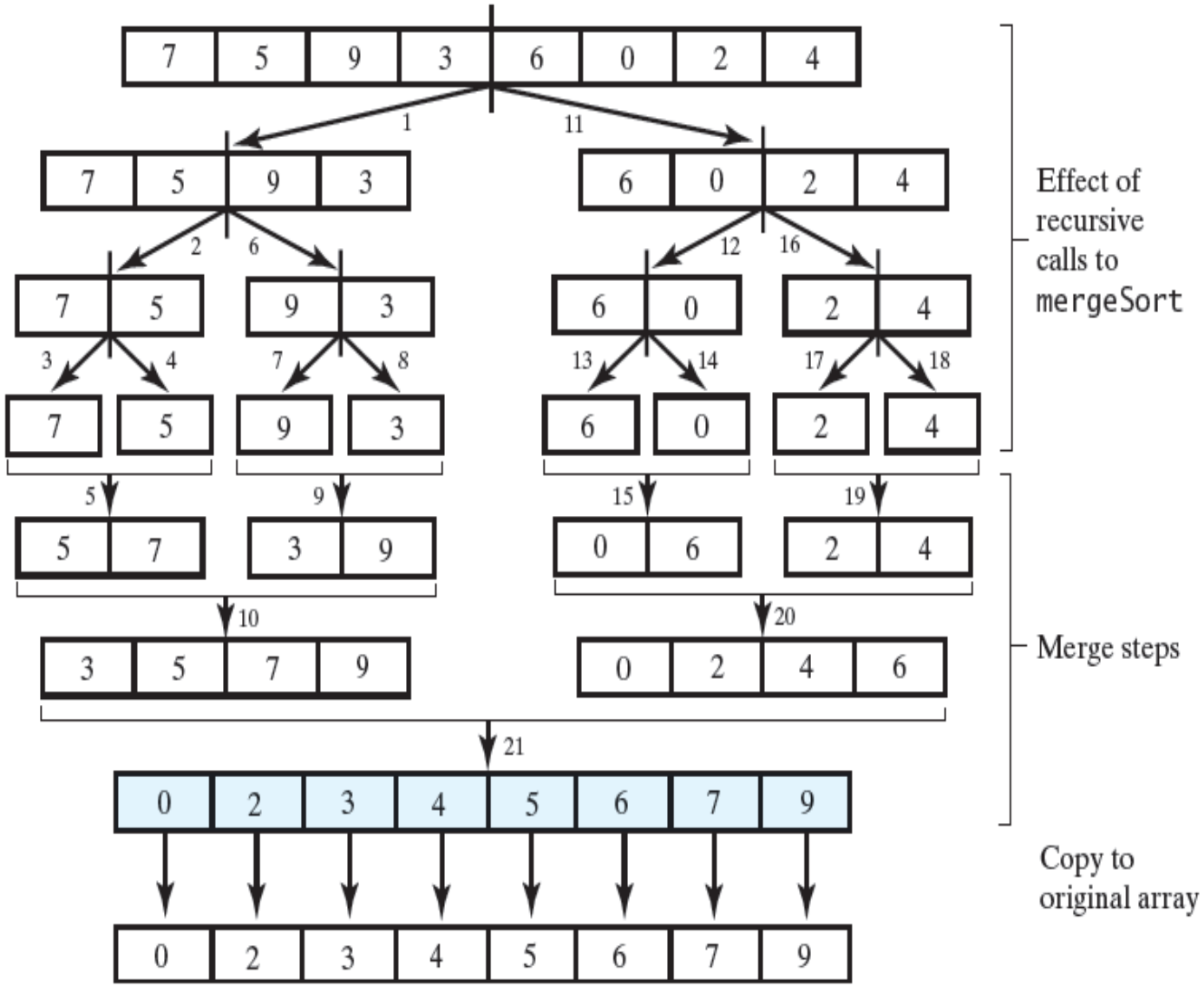
The two sorted sublists are then merged together, picking the smallest elements from the start of each list until all elements have been added back to our full (sorted) list.

Note: MergeSort can take significant space if implemented naively (by creating many sublists)

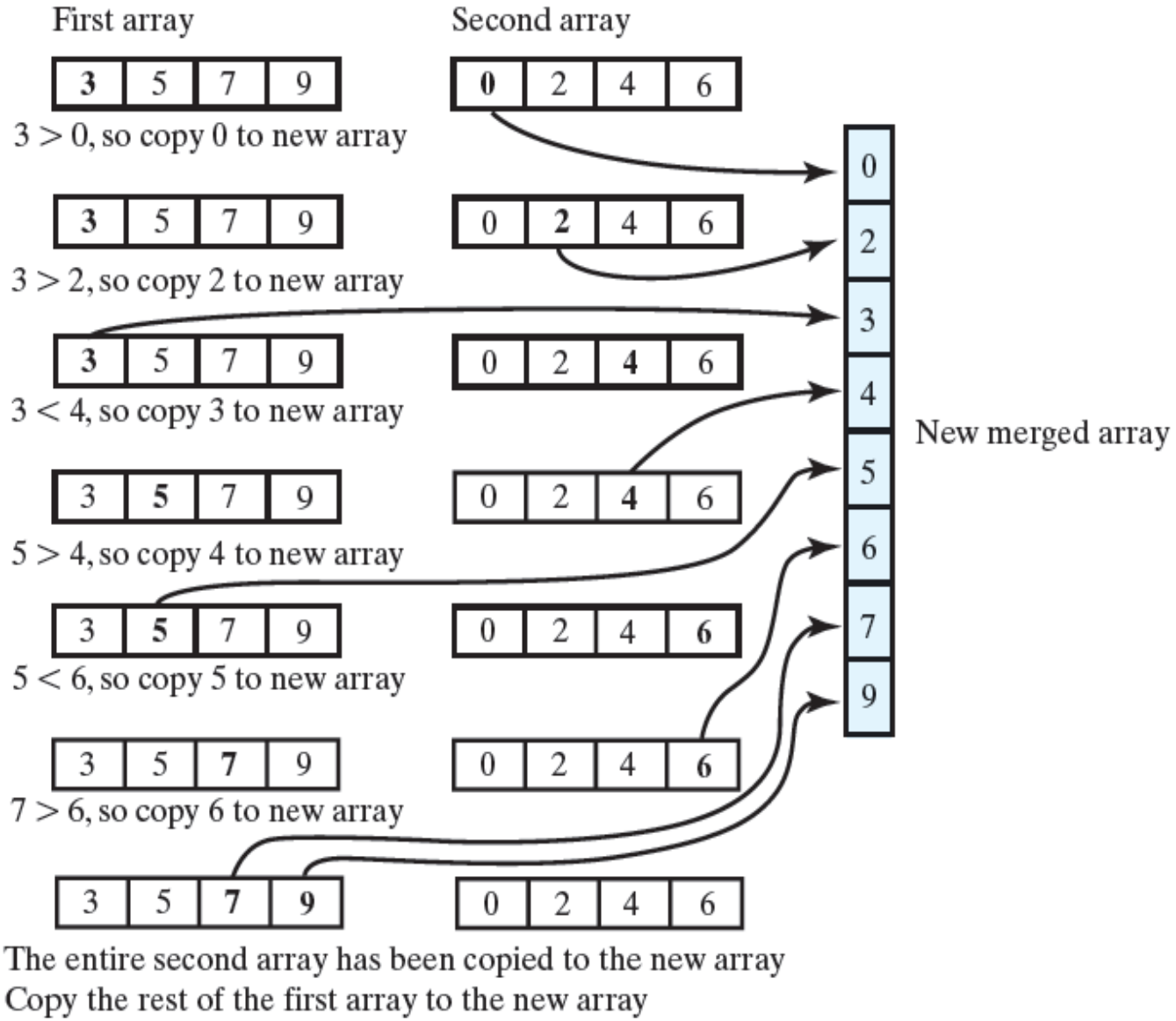
Merge Sort – Overview of the steps



Merge Sort


7 5 9 3 6 0 2 4
1
11
7 5 9 3
6 0 2 4
2
6
12
16
7 5
9 3
6 0
2 4
3
4
7
8
13
14
17
18
7
5
9
3
6
0
2
4
5
9
15
19
5 7
3 9
0 6
2 4
10
20
3 5 7 9
0 2 4 6
21
0 2 3 4 5 6 7 9
0 2 3 4 5 6 7 9
Effect of recursive calls to mergeSort
Merge steps
Copy to original array

Merge Sort – Merging Step



Merge Sort (with primitive types)

```
public void MergeSort(int[] arr, int low, int high)
{
        if (low >= high)
            return;

            // Find the middle point
            int mid = (low+high)/2;

            // Sort first and second halves
            MergeSort(arr, low, mid);
            MergeSort(arr , mid+1, high);

            // Merge the sorted halves
            merge_sorted_arrays(arr, low, mid, high);
}
```

Merge Sort (with object collection)

```
public void MergeSort (ArrayList<Integer> xs){
    // size 1 - already sorted.
    if (xs.size()<=1) { return ;}

    //split into left and right
    int mid = xs.size()/2;
    List<Integer> left  = new ArrayList<Integer>();
    List<Integer> right = new ArrayList<Integer>();

    //put the first half in left.
    for (int i=0; i<mid; i++){  left.add(xs.remove(0)); }
    //put the rest in right.
    while (xs.size()>0) { right.add(xs.remove(0)); }
    …
```

Do you see any inefficiency in this code?

Merge Sort (with generics)

```
public static <T extends Comparable<? super T>> void mergeSort(T[] a)
{
    T[] tmp = (T[]) new Comparable<?>[a.length];
    mergeSort(a, tmp, 0, a.length - 1);
}

private static <T extends Comparable<? super T>>
            void mergeSort(T[] a, T[] tmp, int left, int right)
{
    if( left < right )
    {
        int center = (left + right) / 2;
        mergeSort(a, tmp, left, center);
        mergeSort(a, tmp, center + 1, right);
        merge(a, tmp, left, center + 1, right);
    }
}
```

Merge Sort (with generics)

```
private static <T extends Comparable<? super T>>
    void merge(T[] a, T[] tmp, int left, int right, int rightEnd)
{
   int leftEnd = right - 1;
   int k = left;
   int num = rightEnd - left + 1;
   while(left <= leftEnd && right <= rightEnd){
       if(a[left].compareTo(a[right]) <= 0)
            tmp[k++] = a[left++];
       else
            tmp[k++] = a[right++];
   }
   while(left <= leftEnd)      // Copy rest of first half
       tmp[k++] = a[left++];
   while(right <= rightEnd)    // Copy rest of right half
       tmp[k++] = a[right++];

   for(int i = 0; i < num; i++, rightEnd--)  // Copy tmp back
       a[rightEnd] = tmp[rightEnd];
}
```

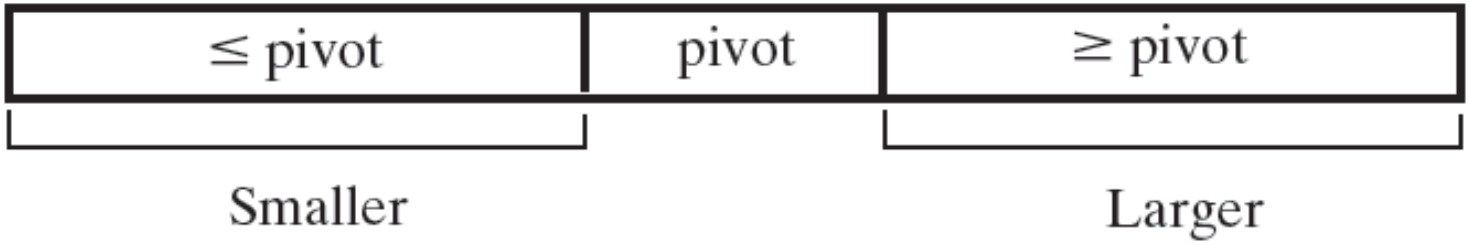
Quicksort

Watch an animation first: https://www.youtube.com/watch?v=tIYMCYooo3c

- Like merge sort, divides arrays into two portions
  - Unlike merge sort, portions not necessarily halves of the array
- One entry called the ***pivot***
  - Pivot goes to position that it will occupy in final sorted array
  - Entries in positions before pivot are smaller than pivot, but **not** in order
  - Entries in positions after pivot are larger than pivot, but **not** in order

Quicksort Algorithm

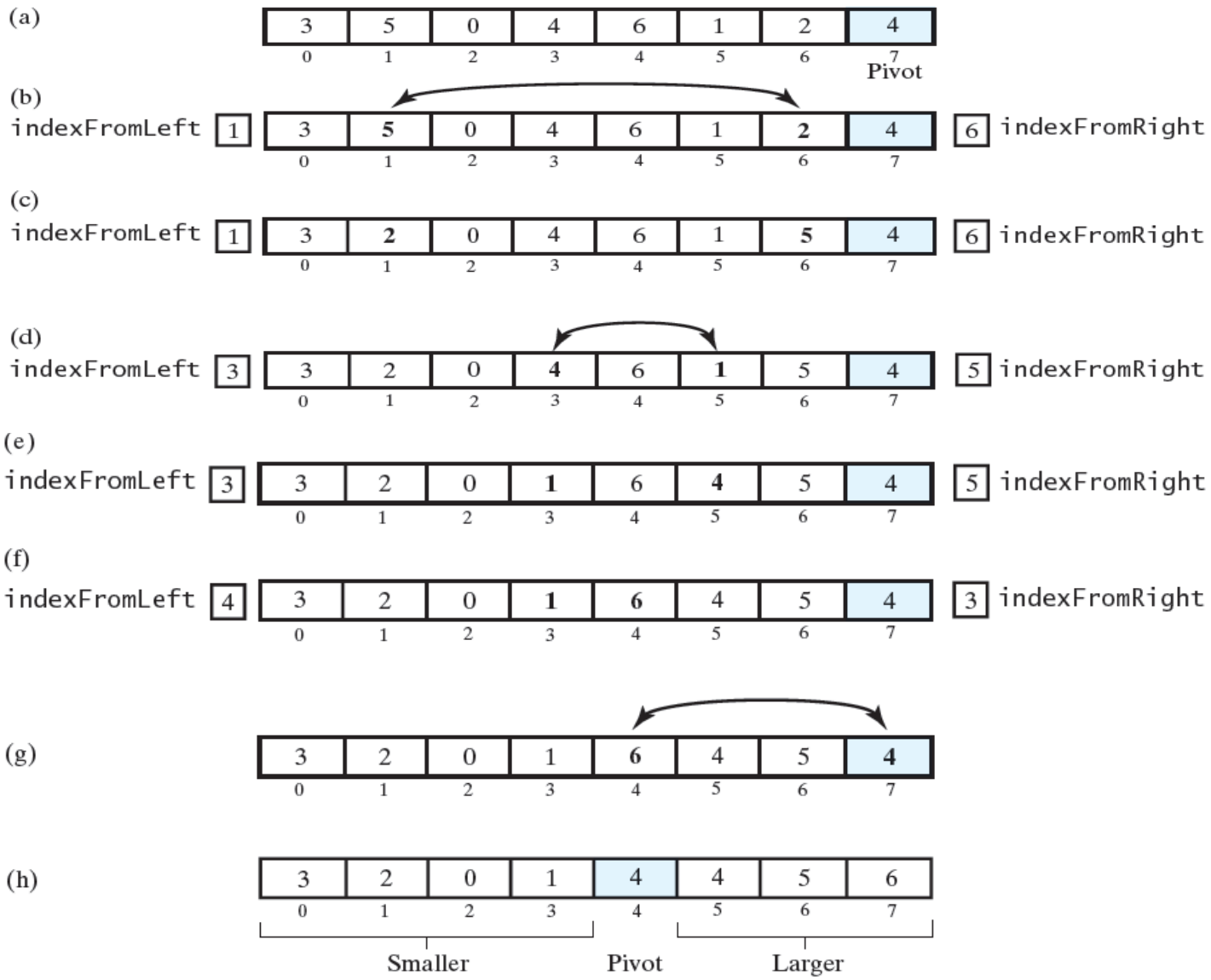
```
quickSort(a[], first, last)
   if first < last
      choose pivot
      quickSort(a, first, pivotIndex-1)
      quickSort(a, pivotIndex+1, last)
```



Quicksort – Partition Algorithm

1. Pick any list element as your pivot.
2. Take two variables to point left and right of the list, excluding pivot.
3. Left *var* points to the lower index. Right *var* points to the higher index.
4. Move all elements which are greater than pivot to the right.
5. Move all elements smaller than the pivot to the left partition.

Quicksort Demo



Quicksort – Pivot Selection

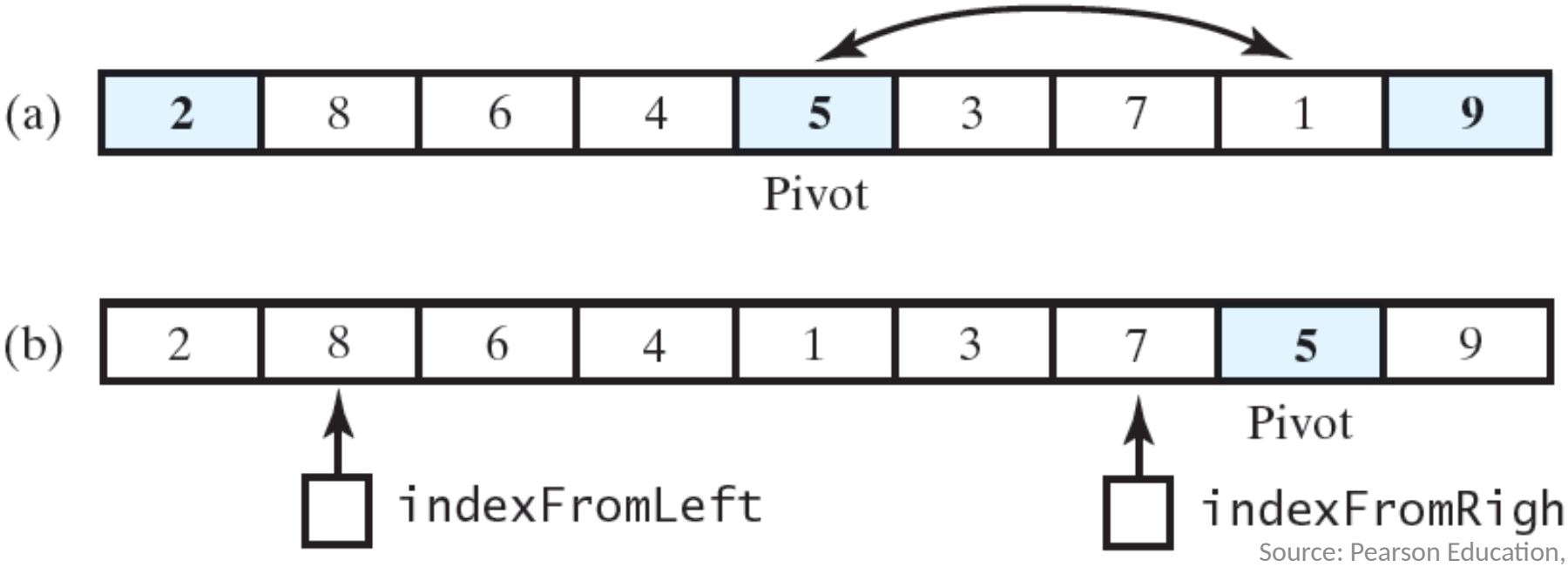
Many alternative approaches:

- Select the first element as pivot
- Select the last element as pivot
- Select a random element as pivot
- Select the median of three as pivot. Typically we use the first entry, the middle entry, and the last entry

Quicksort

Minor modification to partition algorithm

- Pivot swapped with array[last-1]
- indexFromLeft = first + 1
- indexFromLast = last - 2



Source: Pearson Education, Inc

fromleft = 0
fromright = 6

fromleft = 1
fromright = 6

fromleft = 3
fromright = 5

fromleft = 4
fromright = 3

Quicksort (with primitive types)

```
void sort(int[] arr, int low, int high)
{
   if (low < high)
   {
      int pi = partition(arr, low, high);

      sort(arr, low, pi-1);
      sort(arr, pi+1, high);
   }
}
```

```
int partition(int[] arr, int low, int high) {
   int pivot = arr[high];
   int i = (low-1);
   for (int j=low; j<high; j++) {
      if (arr[j] <= pivot)
      {
         i++;
         // swap arr[i] and arr[j]
         int temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
      }
   }

   // swap arr[i+1] and arr[high] (or pivot)
   int temp = arr[i+1];
   arr[i+1] = arr[high];
   arr[high] = temp;

   return i+1;
}
```

Quicksort (with generics)

```
private static <T extends Comparable<? super T>>
        void sortFirstMiddleLast(T[] a, int first, int mid, int last)
{
   order(a, first, mid); // make a[first] <= a[mid]
   order(a, mid, last);  // make a[mid] <= a[last]
   order(a, first, mid); // make a[first] <= a[mid]
} // end sortFirstMiddleLast

private static <T extends Comparable<? super T>>
        void order(T[] a, int i, int j)
{
   if (a[i].compareTo(a[j]) > 0)
      swap(a, i, j);
} // end order

/** Swaps the array entries array[i] and array[j]. */
private static void swap(Object[] array, int i, int j)
{
   Object temp = array[i];
   array[i] = array[j];
   array[j] = temp;
} // end swap
```

Quicksort (with generics)

```
private static <T extends Comparable<? super T>>
        int partition(T[] a, int first, int last)
{
   int mid = (first + last) / 2;
   sortFirstMiddleLast(a, first, mid, last);

   swap(a, mid, last - 1);
   int pivotIndex = last - 1;
   T pivot = a[pivotIndex];

   int indexFromLeft = first + 1;
   int indexFromRight = last - 2;

   boolean done = false;
   while (!done)
   {
         while (a[indexFromLeft].compareTo(pivot) < 0)
            indexFromLeft++;

         while (a[indexFromRight].compareTo(pivot) > 0)
            indexFromRight--;

         assert a[indexFromLeft].compareTo(pivot) >= 0 &&
                a[indexFromRight].compareTo(pivot) <= 0;
```

Quicksort (with generics)

```
        if (indexFromLeft < indexFromRight)
        {
            swap(a, indexFromLeft, indexFromRight);
            indexFromLeft++;
            indexFromRight--;
        }
        else
            done = true;
    } // end while

    // place pivot between Smaller and Larger subarrays
    swap(a, pivotIndex, indexFromLeft);
    pivotIndex = indexFromLeft;

    return pivotIndex;
} // end partition
```

Quicksort (with generics)

```
public static <T extends Comparable<? super T>>
       void quickSort(T[] a, int first, int last)
{
   if (last - first + 1 < MIN_SIZE)
   {
      insertionSort(a, first, last);
   }
   else
   {
      // create the partition: Smaller | Pivot | Larger
      int pivotIndex = partition(a, first, last);

      // sort subarrays Smaller and Larger
      quickSort(a, first, pivotIndex - 1);
      quickSort(a, pivotIndex + 1, last);
   } // end if
} // end quickSort
```

Built-in methods for sorting

Sorting with **`Arrays`** class

```
int[] arr = {-5, 7, 13, 87};
java.util.Arrays.sort(arr);
```

Sorting with **`Collections`** class

```
java.util.List<Integer> arr = new
java.util.ArrayList<Integer>();
arr.add(-5);
arr.add(7);
arr.add(13);
arr.add(87);
java.util.Collections.sort(arr)
```

- Since Java 7, sorting of Objects is using the Timsort algorithm which is a hybrid of merge sort and insertion sort
- Sorting of primitive types is using the Dual-Pivot QuickSort algorithm which is a variation of the quicksort