eC Academy

*Realize Your Dreams*

# AP Computer Science A Review

# Week 9: Algorithm II Sorting/Searching

DR. ERIC CHOU
IEEE SENIOR MEMBER

SECTION 1

# Overview

# Algorithms in McGraw Hills Concept 12

- Swap, Shifting, and Rotation
- Linear Search (Sequential Search)
- Binary Search
- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge Sort
- Quick Sort

SECTION 1

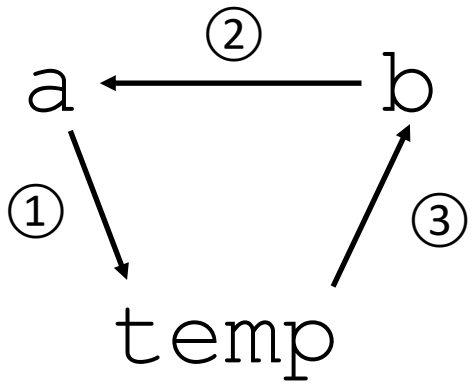
# Swap, Shifting and Rotation

# Application for These Algorithms

- Reverse of Array
- Replacement of Object
- Queue, Priority Queue
- Circular Queue
- Permutation of Data
- Bubble Sort
- Selection Sort
- Quick Sort

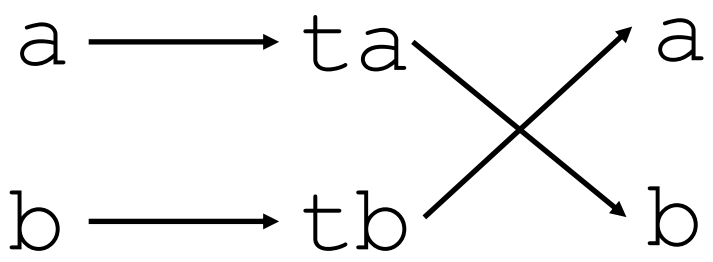
# Swap

```
double temp = a;
a = b;
b = temp;
```




**Rotational Swap**

```
double ta = a;
double tb = b;
b = ta;
a = tb;
```




**Swap by Shuffling Network**

# Swap by Swap Agent

```
37 class Pack<T>{
38     T x;
39     T y;
40     Pack(T a, T b){
41         x = a;
42         y = b;
43     }
44
45     Pack swap(){
46         return new Pack(y, x);
47     }
48 }
```

- Re-usable
- Generic
- Small amount of memory overhead

```java
public class Swap{
    public static void main(String[] args){
        System.out.println("\fBefore Rotational Swap: ");
        int a = 3;
        int b = 4;
        System.out.println("A="+a+" B="+b);
        int temp = a;
        a = b;
        b = temp;
        System.out.println("After Rotational Swap:");
        System.out.println("A="+a+" B="+b);

        System.out.println("\nBefore Shuffle Swap: ");
        a = 3;
        b = 4;
        System.out.println("A="+a+" B="+b);
        int ta = a;
        int tb = b;
        a = tb;
        b = ta;
        System.out.println("After Shuffle Swap:");
        System.out.println("A="+a+" B="+b);

        System.out.println("\nBefore Swap Agent Swap: ");
        a = 3;
        b = 4;
        System.out.println("A="+a+" B="+b);
        Pack<Integer> p = new Pack<Integer>(a, b);
        p=p.swap();
        a = p.x;
        b = p.y;
        System.out.println("After Shuffle Swap:");
        System.out.println("A="+a+" B="+b);
    }
}
```

BlueJ: Terminal Wi...

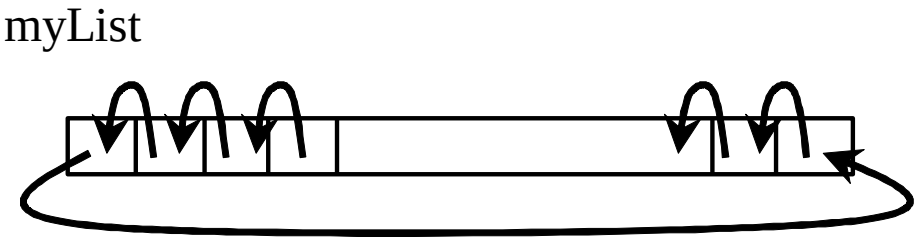
Options

```
Before Rotational Swap:
A=3 B=4
After Rotational Swap:
A=4 B=3

Before Shuffle Swap:
A=3 B=4
After Shuffle Swap:
A=4 B=3

Before Swap Agent Swap:
A=3 B=4
After Shuffle Swap:
A=4 B=3
```

Can only enter input while yo

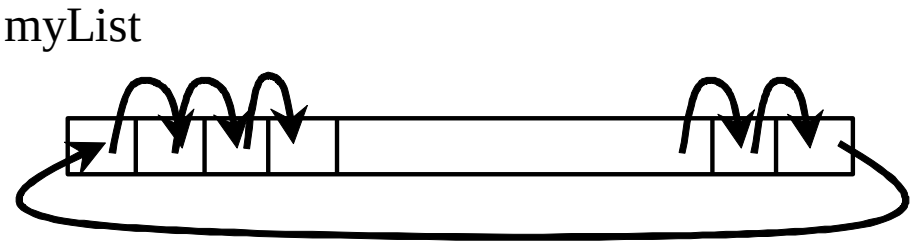
# Shifting Elements (Left Shifting)

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



# Shifting Elements (Right Shifting)

```
double temp = myList[myList.length-1]; // Retain the last element

// Shift elements left
for (int i = myList.length-2; i >=0; i--) {
  myList[i + 1] = myList[i];
}

// Move the last element to fill in the first position
myList[0] = temp;
```



SECTION 1

# Linear Search

# Linear Search

- Involves going through each element of a collection until the appropriate value is found.
- This is often implemented using a **simple for loop**, where you begin at index 0 and count up to the last index in the collection.
- This is the simplest search method, and becomes less effective if the collection is very large.

# Linear Search (Sequential Search)

```
public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
        if (key == list[i])
            return i;
    }
    return -1;
}
```

SECTION 1

# Binary Search

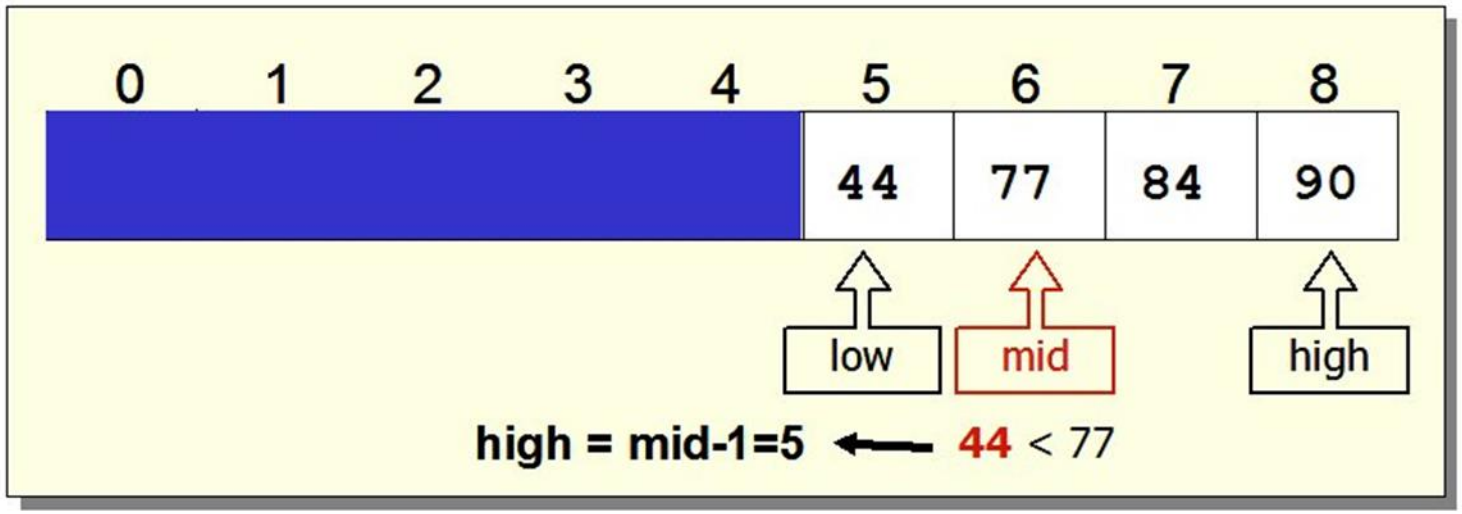
# Binary Search

- Involves going through a collection and comparing the middle index against the value. This requires a sorted collection in order to work.
- For Example: You are trying to find the number 12 in an array of 100 integers, with values equal to the index+1 (1,2,3,4,5,..,99,100). Notice that this array is sorted with the smallest values on the left and the largest at the right. You would compare the middle value (50) against the search value (12). Since 50>12, you would exclude all the values 50 to 100 since they are also greater than the search value. You would then check 25 against the search value since that is the new middle between 49 and 1. Once again this value is larger than the search term and so all number 25 to 49 are excluded. This process continues until you finally arrive with 12 as the center value.

| | low | high | mid |
|---|---|---|---|
| #1 | 0 | 8 | 4 |
| #2 | 5 | 8 | 6 |

search( 44 )

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



# Binary Search

```
1  public class BinarySearch {
2    /** Use binary search to find the key in the list */
3    public static int binarySearch(int[] list, int key) {
4      int low = 0;
5      int high = list.length - 1;
6
7      while (high >= low) {
8        int mid = (low + high) / 2;
9        if (key < list[mid])
10         high = mid - 1;
11       else if (key == list[mid])
12         return mid;
13       else
14         low = mid + 1;
15     }
16     return -low - 1; // Now high < low
17   }
```

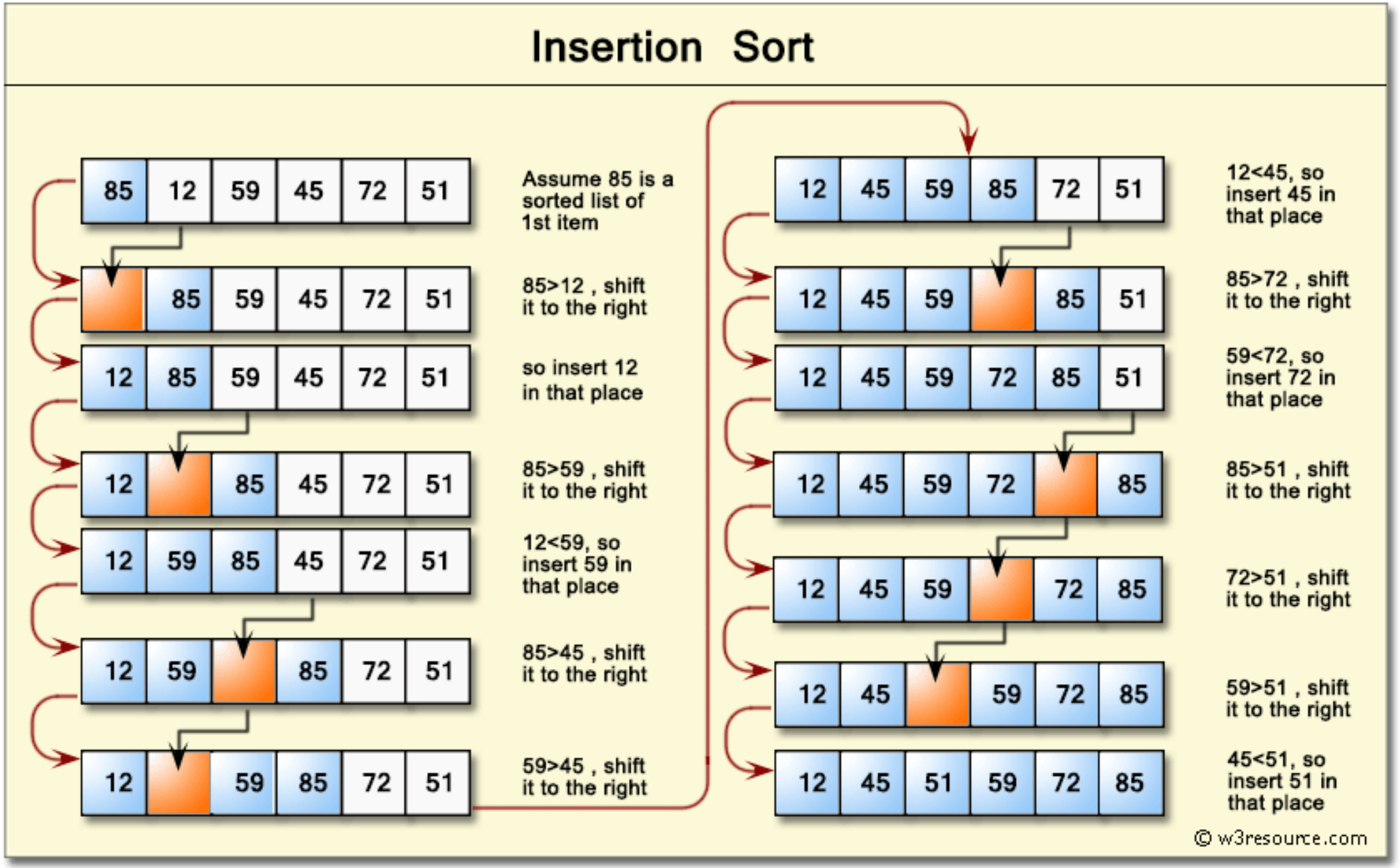
mid = (low+high)/2
= (0+length-1)/2
= (lengh-1)/2

length = 10, mid = 4 (2nd half longer)
length = 9, mid = 4 (balance)

**Mid-point belong to lower half.**

SECTION 1

# Insertion Sort



```java
public class InsertionSort {
  /** The method for sorting the numbers */
  public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
      /** insert list[i] into a sorted sublist list[0..i-1] so that
          list[0..i] is sorted. */
      int currentElement = list[i];
      int k;
      for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
        list[k + 1] = list[k];
      }
      // Insert the current element into list[k+1]
      list[k + 1] = currentElement;
    }
  }

  /** A test method */
  public static void main(String[] args) {
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    insertionSort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }
}
```

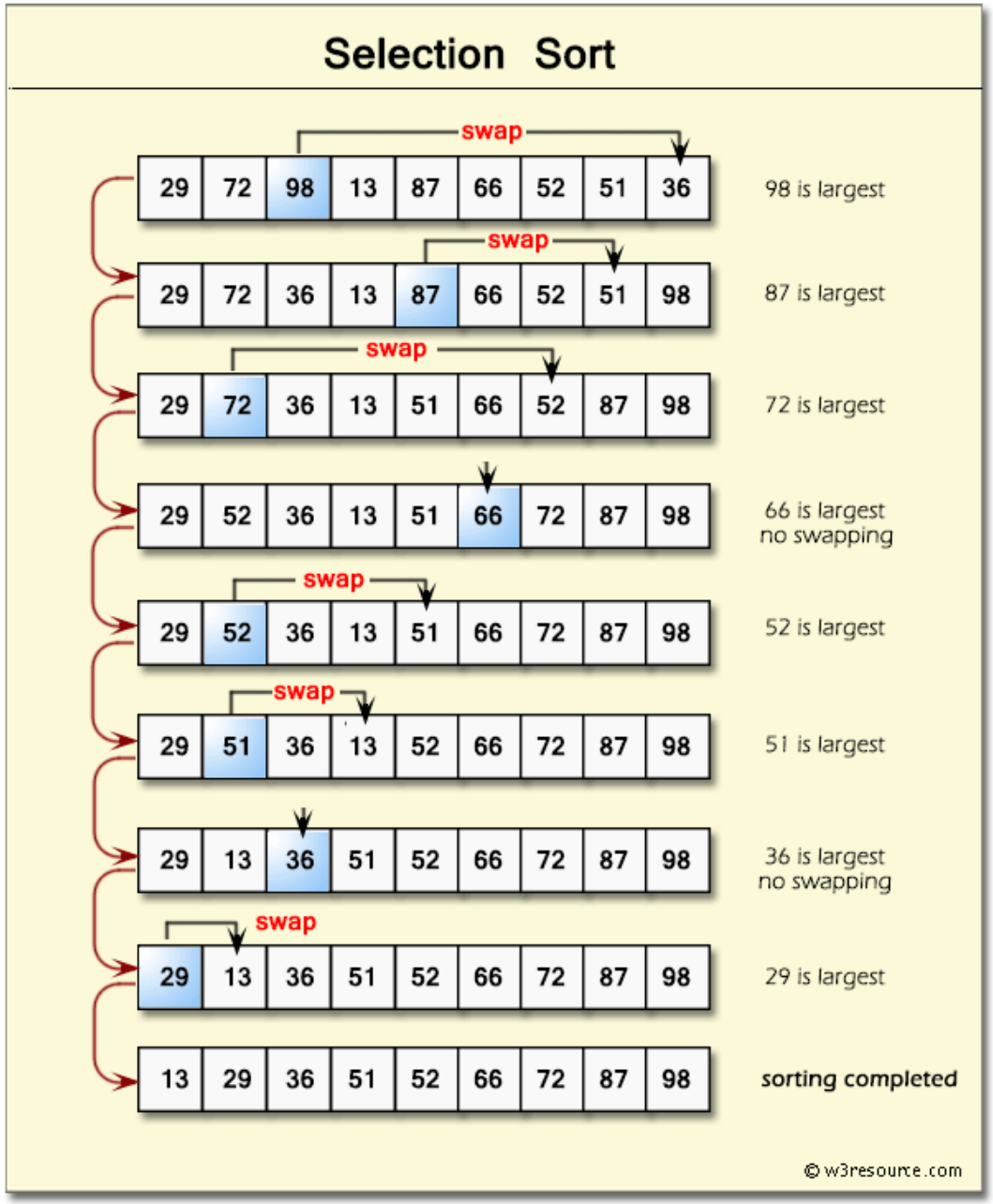
# Insertion Sort

SECTION 1

# Selection Sort

# Selection Sort

- Selection sort is an iterative sort algorithm that uses a "search and swap" approach to sort a collection. For each pass through the collection, the algorithm finds the smallest element to be sorted and swaps it with the first unsorted element in the collection.
- The algorithm continues in this manner, finding the next smallest element in the collection and swapping it with the next unsorted element. Finally, when just two unsorted elements remain, they are compared (and if necessary, swapped) and the sort is complete.



```java
public class SelectionSort {
  /** The method for sorting the numbers */
  public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length - 1; i++) {
      // Find the minimum in the list[i..list.length-1]
      double currentMin = list[i];
      int currentMinIndex = i;
      for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }
      // Swap list[i] with list[currentMinIndex] if necessary;
      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
  }
```

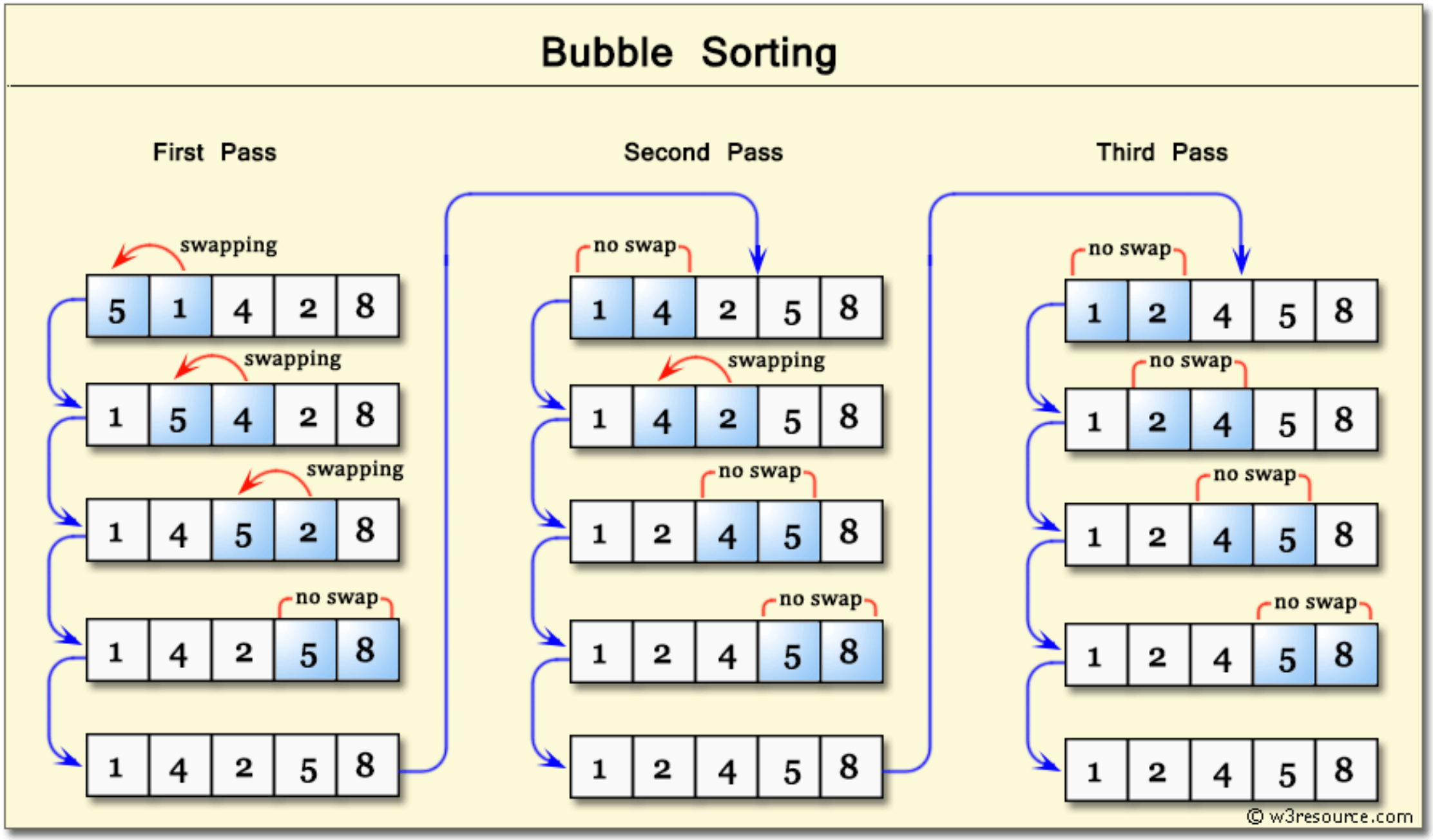
# Selection Sort

SECTION 1

# Bubble Sort

# Bubble sort

- Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.
- Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.



```java
public class BubbleSort {
  /** Bubble sort method */
  public static void bubbleSort(int[] list) {
    boolean needNextPass = true;

    for (int k = 1; k < list.length && needNextPass; k++) {
      // Array may be sorted and next pass not needed
      needNextPass = false;
      for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1]) {
          // Swap list[i] with list[i + 1]
          int temp = list[i];
          list[i] = list[i + 1];
          list[i + 1] = temp;

          needNextPass = true; // Next pass still needed
        }
      }
    }
  }
```

# Bubble Sort

SECTION 1

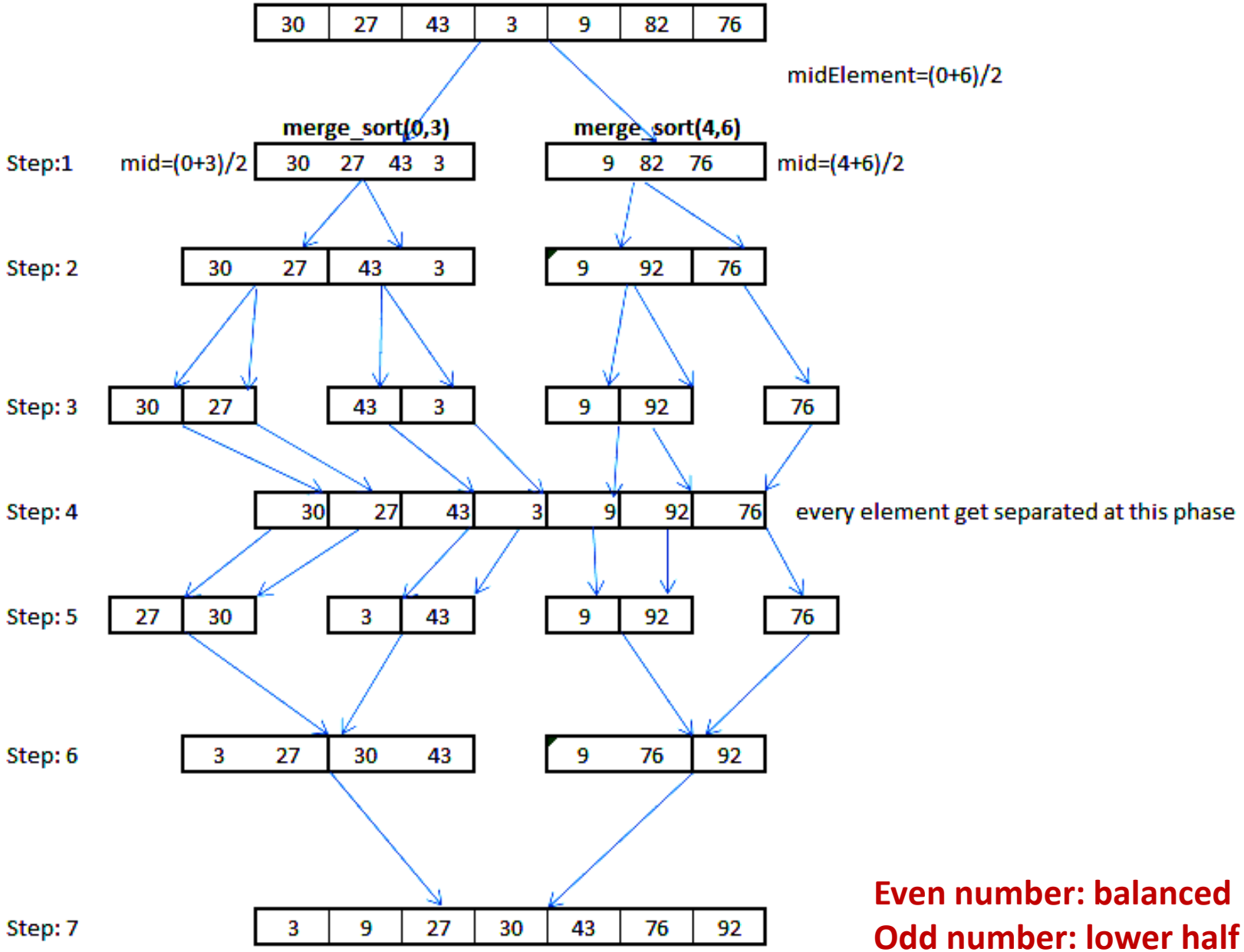
# Merge Sort

# Merge Sort

- Mergesort is a recursive algorithm that uses a "divide and conquer" approach to sorting collections. Each time the algorithm is called, the algorithm checks to see if there is more than one element in the collection, and if there is, the collection is "broken" into two halves.
- If there is even number of elements, the **halves** are equal, but if there is an odd number of elements, the **left half** will contain one more element than the right half.
- At this point, the algorithm uses recursion, calling itself to first mergesort the left half of the collection, then mergesort the right half. When the algorithm calls itself to mergesort one of the halves, it again "breaks" the half into two halves, then calls itself to sort each half. This process is repeated until the entire collection is "broken" into individual elements, or sub-collections of length 1. When the method calls itself to sort one of these, the initial test that sees if there is more than one element in the sub-collection fails, since there is only one element in the sub-collection.

# Merge Sort

- This is where the recursion stops, and the algorithm returns to sorting the half by comparing the two adjacent elements, sorting them, and then recombining them into a sorted sub-collection.
- This process continues until all of the individual elements have been sorted and recombined into sorted sub-collections.
- Then the sub-collections themselves are compared, sorted, and recombined.
- This process continues until the sub-collections have been recombined to form two sorted halves. Finally, the left and right halves are compared to each other, sorted, and recombined to create the final, fully sorted collection.




**Even number: balanced**
**Odd number: lower half has one more.**

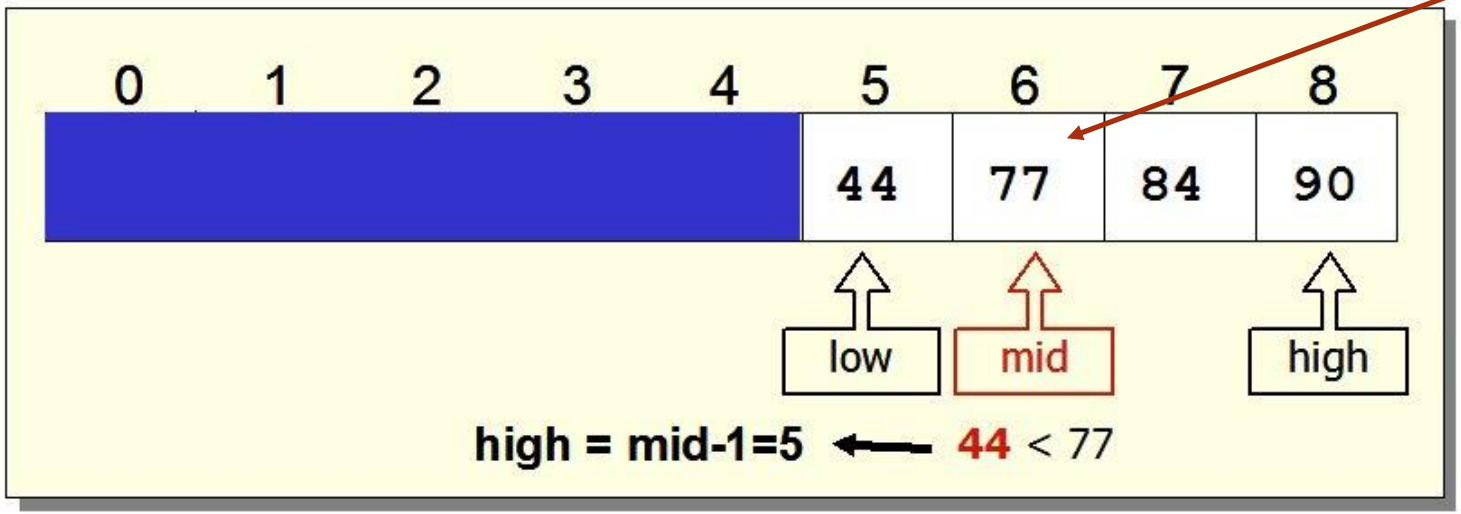
# We use the same low, mid, high system as the Binary Search

| | low | high | mid |
|---|---|---|---|
| #1 | 0 | 8 | 4 |
| #2 | 5 | 8 | 6 |

search( 44 )

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

mid belongs to first half



mid belongs to first half

```java
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    // Merge sort the first half
    int low = 0, high = list.length-1;
    int mid = (low+high)/2;
    int[] firstHalf = new int[mid+1];
    System.arraycopy(list, 0, firstHalf, 0, mid+1);
    mergeSort(firstHalf);

    // Merge sort the second half
    int secondHalfLength = list.length - (mid+1);
    int[] secondHalf = new int[secondHalfLength];
    System.arraycopy(list, mid+1,
      secondHalf, 0, secondHalfLength);
    mergeSort(secondHalf);

    // Merge firstHalf with secondHalf into list
    merge(firstHalf, secondHalf, list);
  }
}
```

```java
/** Merge two sorted lists */
public static void merge(int[] list1, int[] list2, int[] temp) {
  int current1 = 0; // Current index in list1
  int current2 = 0; // Current index in list2
  int current3 = 0; // Current index in temp

  while (current1 < list1.length && current2 < list2.length) {
    if (list1[current1] < list2[current2])
      temp[current3++] = list1[current1++];
    else
      temp[current3++] = list2[current2++];
  }

  while (current1 < list1.length)
    temp[current3++] = list1[current1++];

  while (current2 < list2.length)
    temp[current3++] = list2[current2++];
}
```

SECTION 1

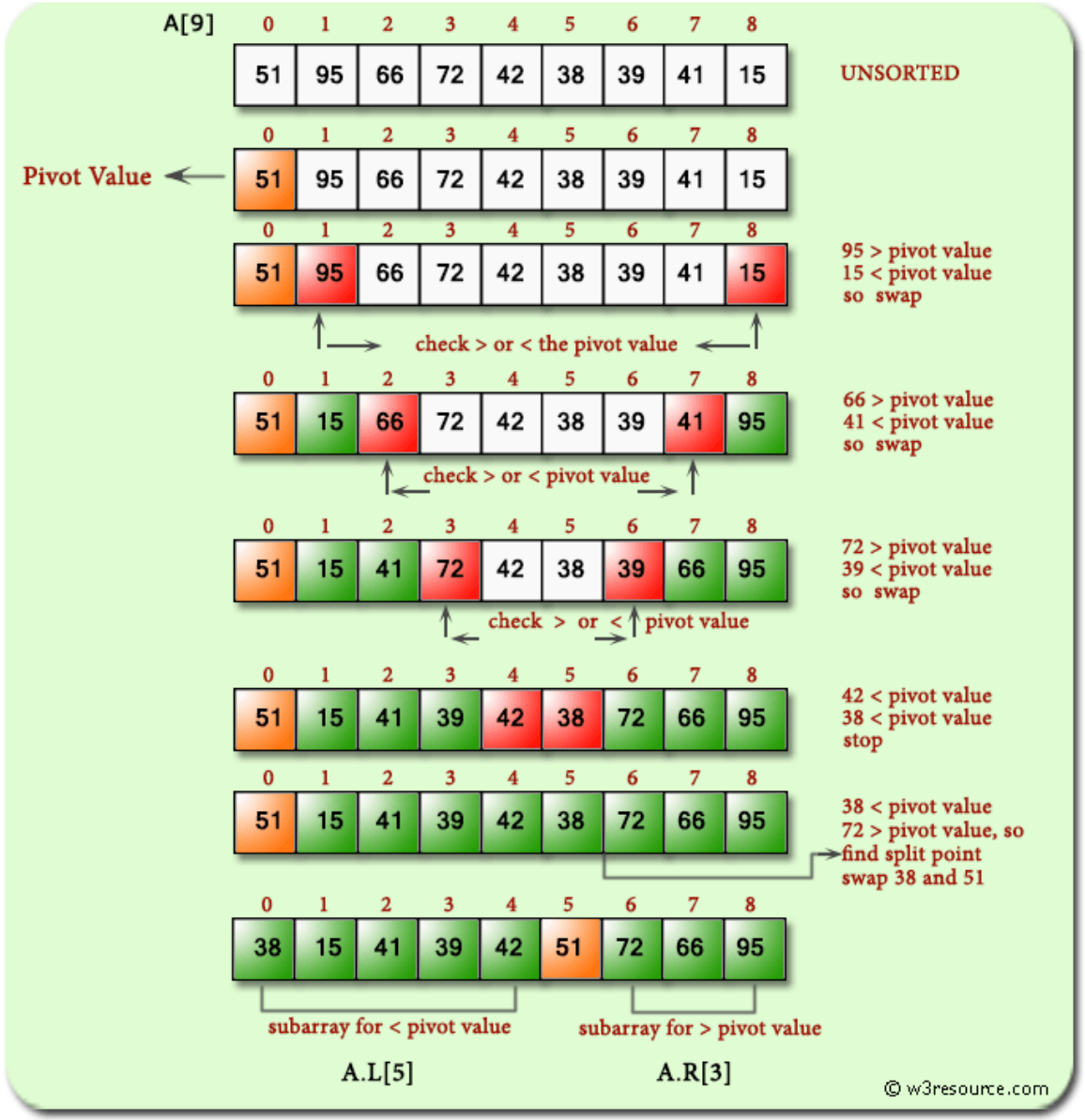
# Quick Sort

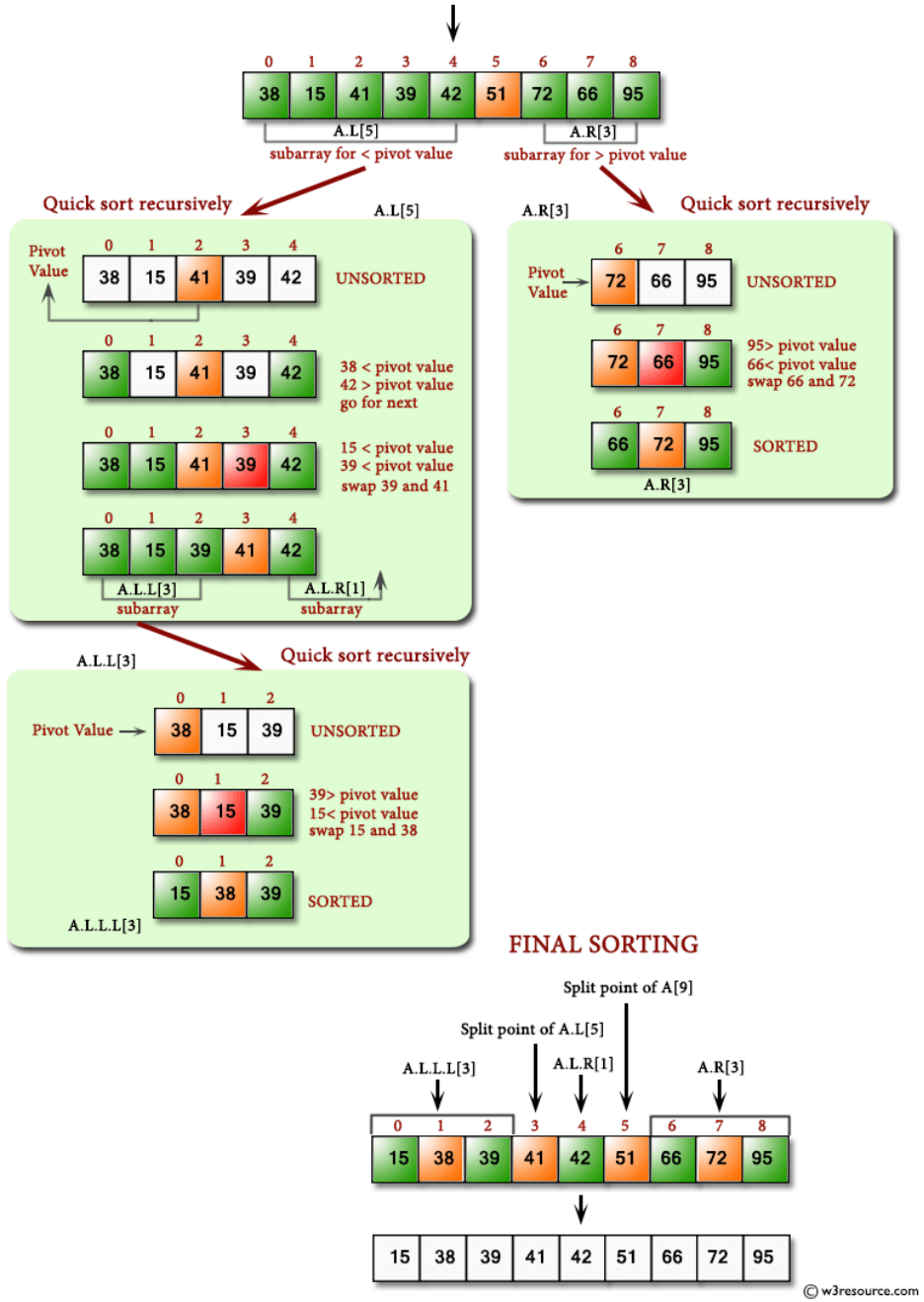
# Quick Sort

## Suitable for ArrayList and Recursive

- Quicksort is, on average, the fastest sorting algorithm for sorting collections with a large number of elements.
- Quicksort is recursive and also uses a "divide and conquer" approach to sorting. This algorithm starts by partitioning the collection, selecting a pivot point, which is usually either the first element in the collection or an element selected at random, then moving all elements less than the pivot point value to the left of the pivot point, and all elements greater than or equal to the pivot point value to the right of the pivot point.
- The algorithm then uses recursion, splitting the array into two halves and calling itself to quicksort each half.
- Finally, the sort is complete when every element has been moved to its correct location.

# Quick Sort




## FINAL SORTING

```java
public static void quickSort(int[] list) {
  quickSort(list, 0, list.length - 1);
}

private static void quickSort(int[] list, int first, int last) {
  if (last > first) {
    int pivotIndex = partition(list, first, last);
    quickSort(list, first, pivotIndex - 1);
    quickSort(list, pivotIndex + 1, last);
  }
}
```

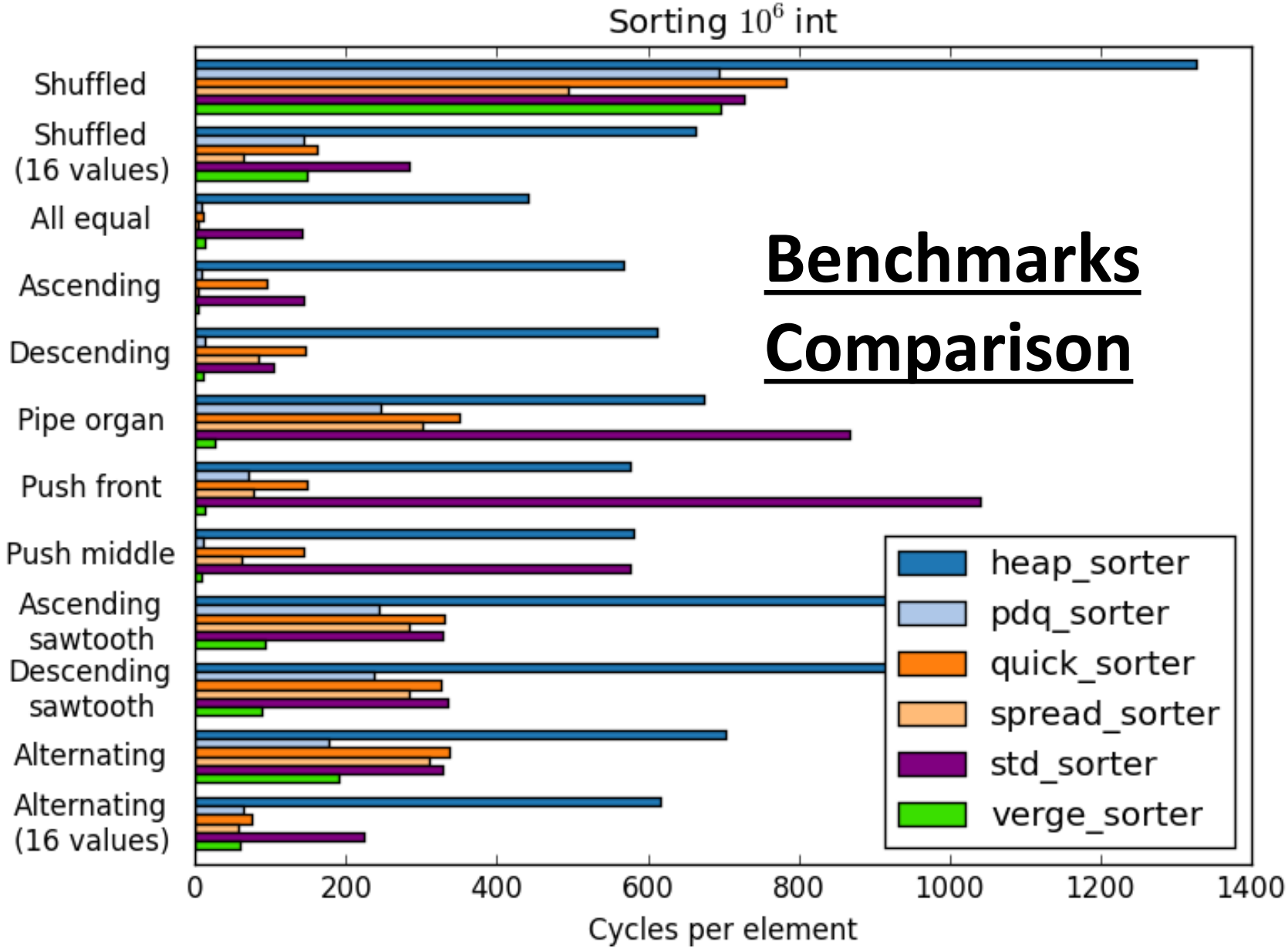
```java
/** Partition the array list[first..last] */
private static int partition(int[] list, int first, int last) {
  int pivot = list[first]; // Choose the first element as the pivot
  int low = first + 1; // Index for forward search
  int high = last; // Index for backward search

  while (high > low) {
    // Search forward from left
    while (low <= high && list[low] <= pivot) low++;
    // Search backward from right
    while (low <= high && list[high] > pivot) high--;
    // Swap two elements in the list
    if (high > low) {
      int temp = list[high];
      list[high] = list[low];
      list[low] = temp;
    }
  }

  while (high > first && list[high] >= pivot) high--;

  // Swap pivot with list[high]
  if (pivot > list[high]) {
    list[first] = list[high];
    list[high] = pivot;
    return high;
  }
  else { return first;
  }
}
```

SECTION 1

# Comparison of Sorting Algorithms

# Time Complexity Comparison

| Sorting algorithm | Time complexity | | | Space complexity | Stability |
|---|---|---|---|---|---|
| | Average case | Best case | Worst case | | |
| Bubble sort[2] | $O(N^2)$ | $O(N)$ | $O(N^2)$ | $O(1)$ | Yes |
| Insertion sort[3,4] | $O(N^2)$ | $O(N)$ | $O(N^2)$ | $O(1)$ | Yes |
| Selection sort[5,6] | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ | $O(1)$ | No |
| Merge sort[7] | $O(N \log N)$ | $O(N \log N)$ | $O(N \log N)$ | $O(N)$ | Yes |
| Heap sort[1,5] | $O(N \log N)$ | $O(N \log N)$ | $O(N \log N)$ | $O(1)$ | No |
| Quick sort[8,9] | $O(N \log N)$ | $O(N \log N)$ | $O(N^2)$ | $O(N \log N)$ | No |
| Counting sort[10,11] | $O(N)$ | $O(N)$ | $O(N^2)$ | $O(N)$ | Yes |
| Radix sort[3] | $O(N)$ | $O(N)$ | $O(N)$ | $O(N)$ | No |
| Bucket sort[3] | $O(N)$ | $O(N)$ | $O(N^2)$ | $O(N)$ | Yes |




**Benchmarks Comparison**

**Benchmarks Comparison**

SECTION 1

# Software Engineering

# Software Engineering

- Software Engineering is the study of designing, developing, and maintaining software.
- Over the years, many different software development models have been designed that help developers create quality software.

# Software Development Models

- **Prototyping** – an approximation of a final system is built, tested, and reworked until it is acceptable. The complete system is then developed from this prototype.
- **Incremental Development** – The software is designed, implemented, and tested a little bit at a time until the product is finished.
- **Rapid Application Development** – the user is actively involved in the evaluation of the product and modifications are made immediately as problems are found. Radical changes in the system are likely at any moment in the process.

# Software Development Models

- **Agile Software Development** – the developers offer frequent releases of the software to the customer and new requirements are generated by the users. Short development cycles are common in this type of development.
- **Waterfall Model** – The progress of development is sequential and flows downward like a waterfall. Steps include conception, initiation, analysis, design, construction, testing, implementation, and maintenance.

# Program Specifications

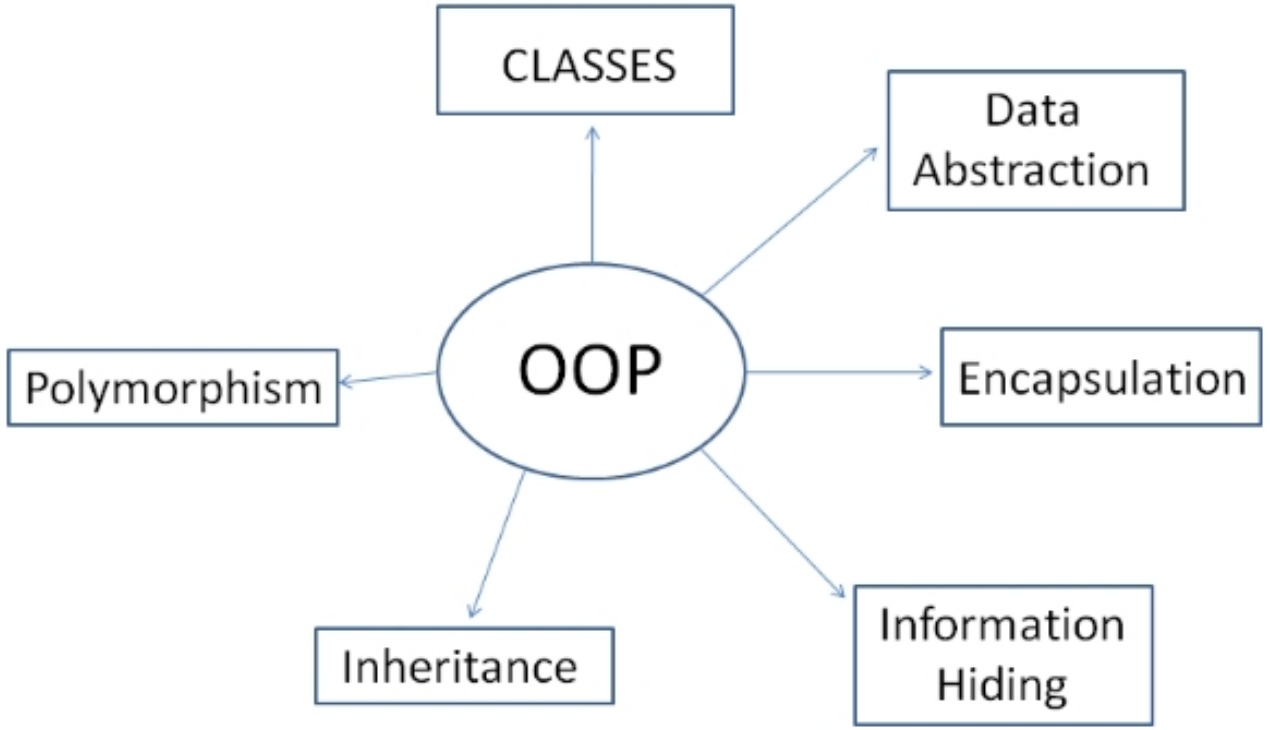
- A program specification describes the results that a program is expected to produce -- its primary purpose is to be understood not executed. Specifications provide the foundation for programming methodology.
- A specification is a **technical contract** between a programmer and his/her client and is intended to provide them with a mutual understanding of a program.
- A client uses the specification to guide his/her use of the program; a programmer uses the specification to guide his/her construction of the program. A complex specification may engender sub specifications, each describing a sub component of the program.
- The construction of these sub components may then be delegated to other programmers so that a programmer at one level becomes also a client at another.

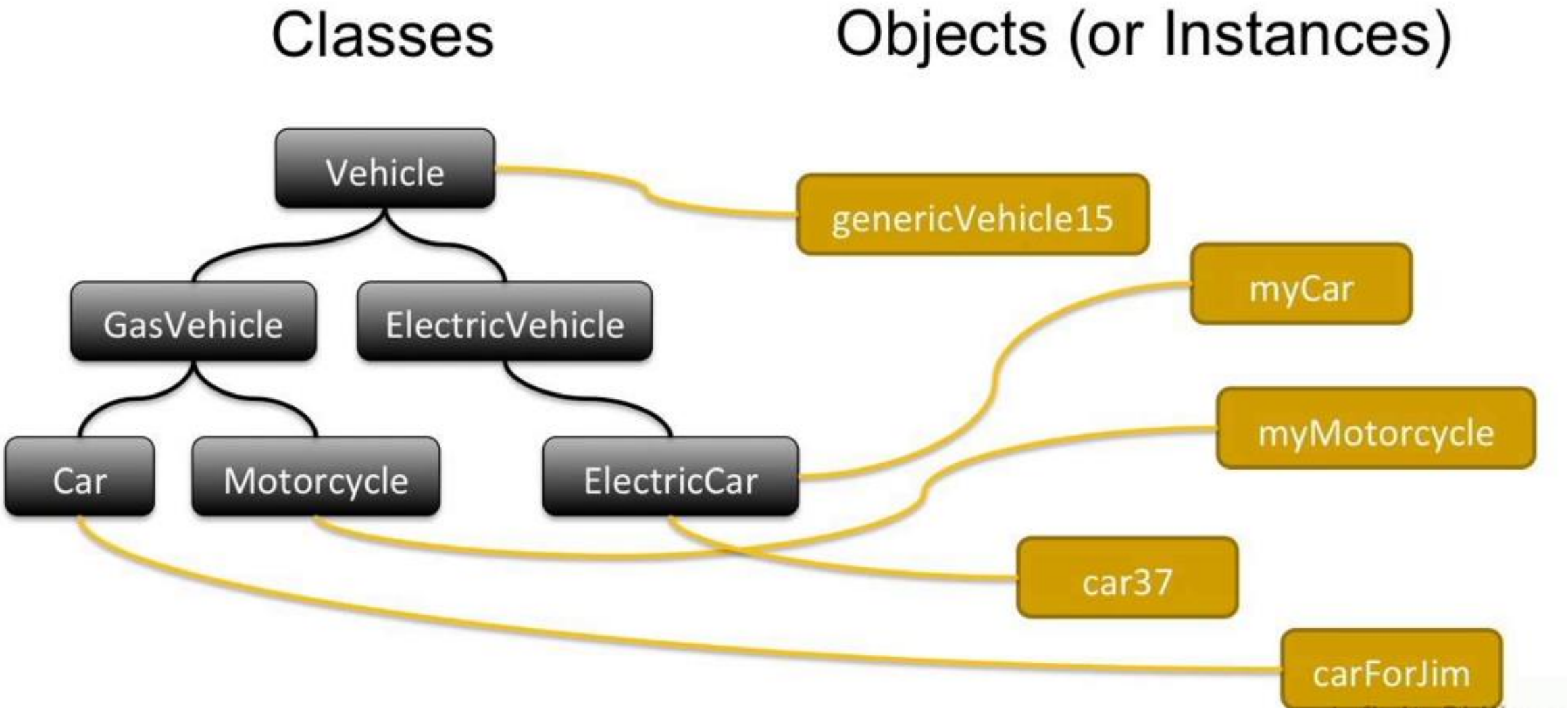
# Design Class Hierarchy

- Top-down and bottom-up are two ways of approaching class hierarchy design.  Top-down is also referred to as functional decomposition. The two designs are different only in their approach to the problem.
- Top-down design starts with the big picture, whereas a bottom-up design starts with the details. They each work toward the other, but where they begin is different.

# Top Down and Bottom Up Design

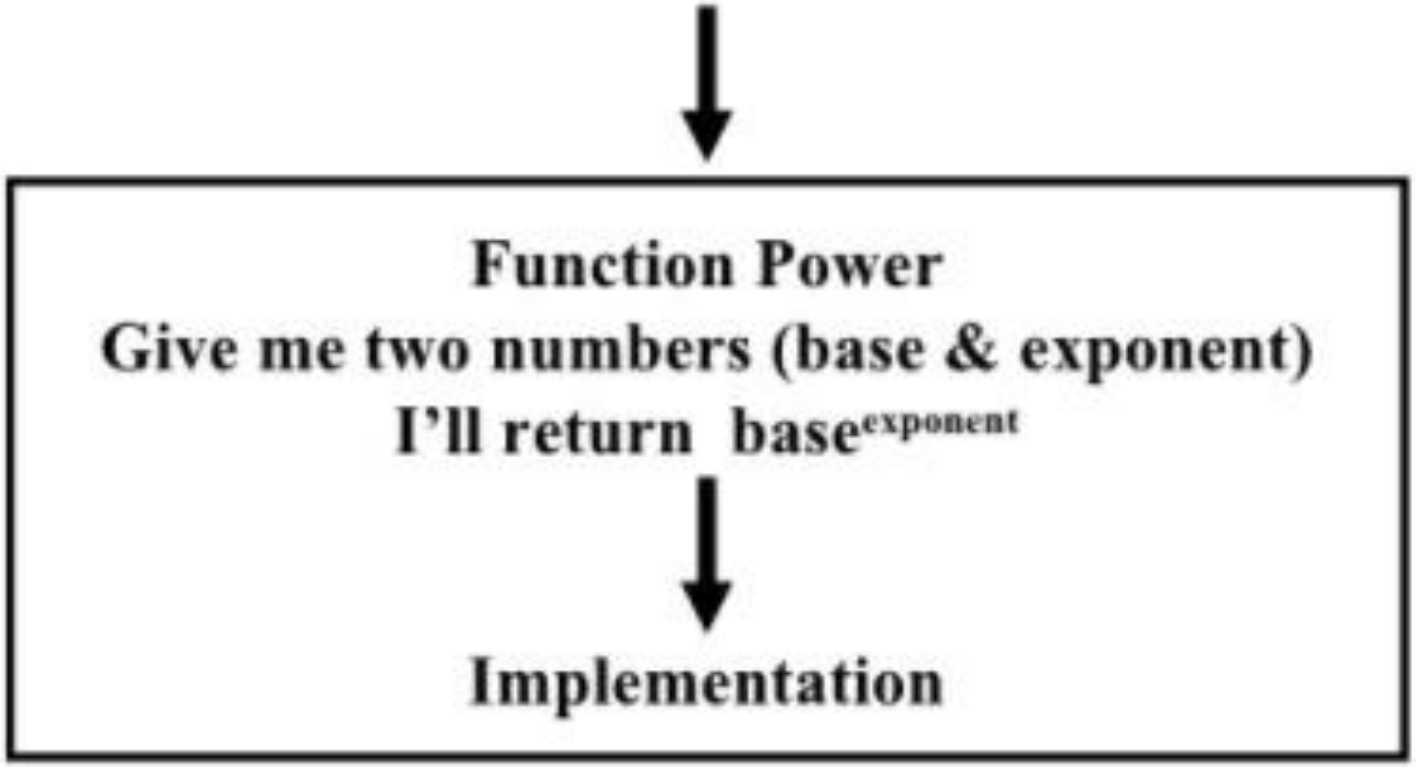
# Object-Oriented Programming

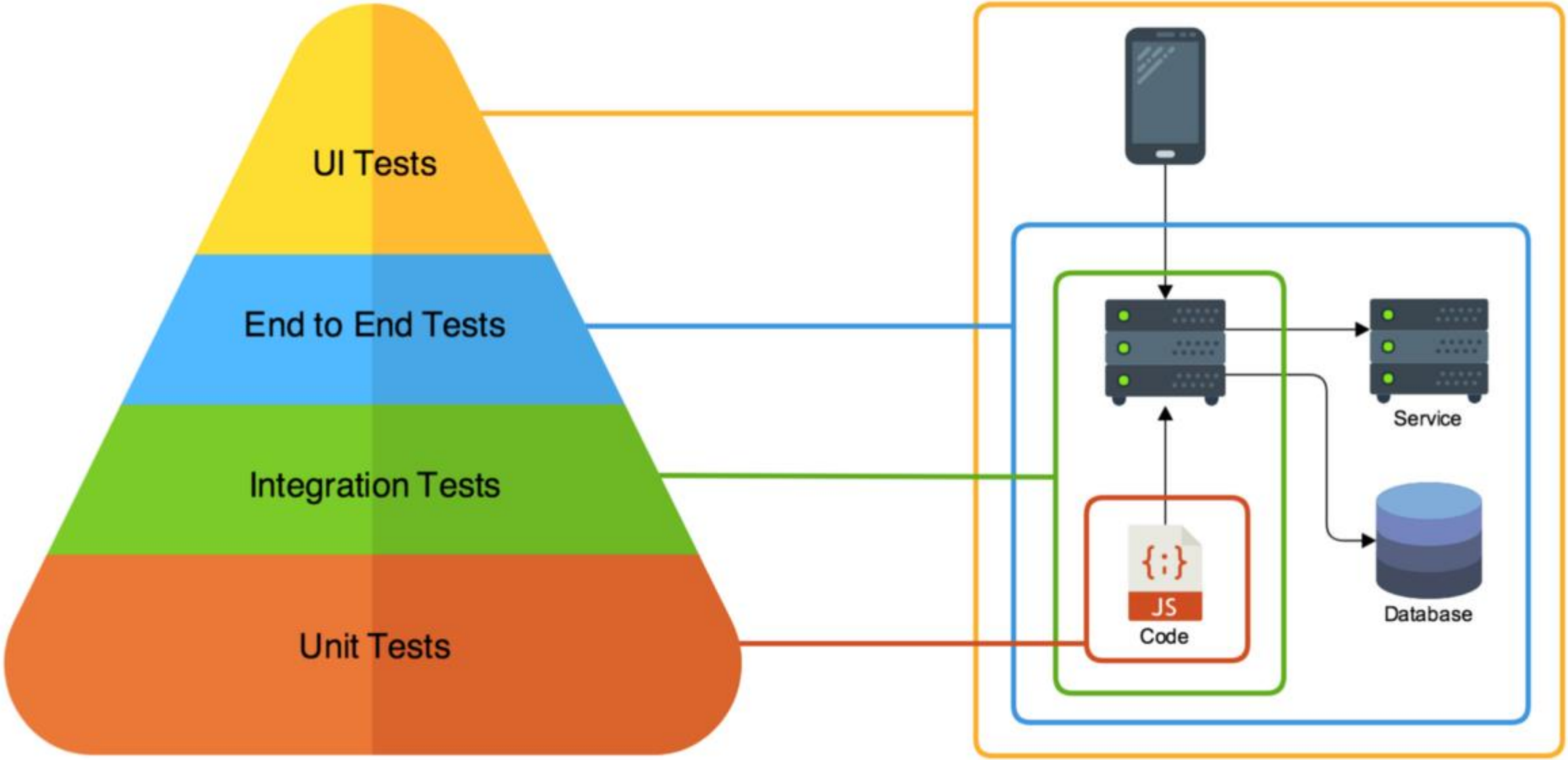



Classes
Objects (or Instances)
Vehicle
GasVehicle
ElectricVehicle
Car
Motorcycle
ElectricCar
genericVehicle15
myCar
myMotorcycle
car37
carForJim

# Procedural Abstraction

- Procedural Abstractions organize instructions.




UI Tests
End to End Tests
Integration Tests
Unit Tests
Service
JS
Code
Database