AP Computer Science A

Java Programming Essentials [Ver.4.0]

Unit 4: Data Collections

CHAPTER 20: SORTING AND SEARCHING ALGORITHMS

DR. ERIC CHOU
IEEE SENIOR MEMBER

AP Computer Science Curriculum

- Searching Algorithms (T4.14)
- Sorting Algorithms (T4.15)

Objectives

- Euclid’s Algorithm
- Finding Prime Numbers
- Linear Search
- Binary Search

Objectives

- Array.sort()
- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge Sort
- Quick Sort

Standard Algorithms

Lecture 1

Euclid's Algorithm

Given two numbers not prime to one another, to find their greatest common measure.

- What Euclid called "common measure" is termed nowadays a **common factor or a common divisor**.

- Euclid VII.2 then offers an **algorithm** for finding the **greatest common divisor** (gcd) of two integers. Not surprisingly, the algorithm bears Euclid's name.

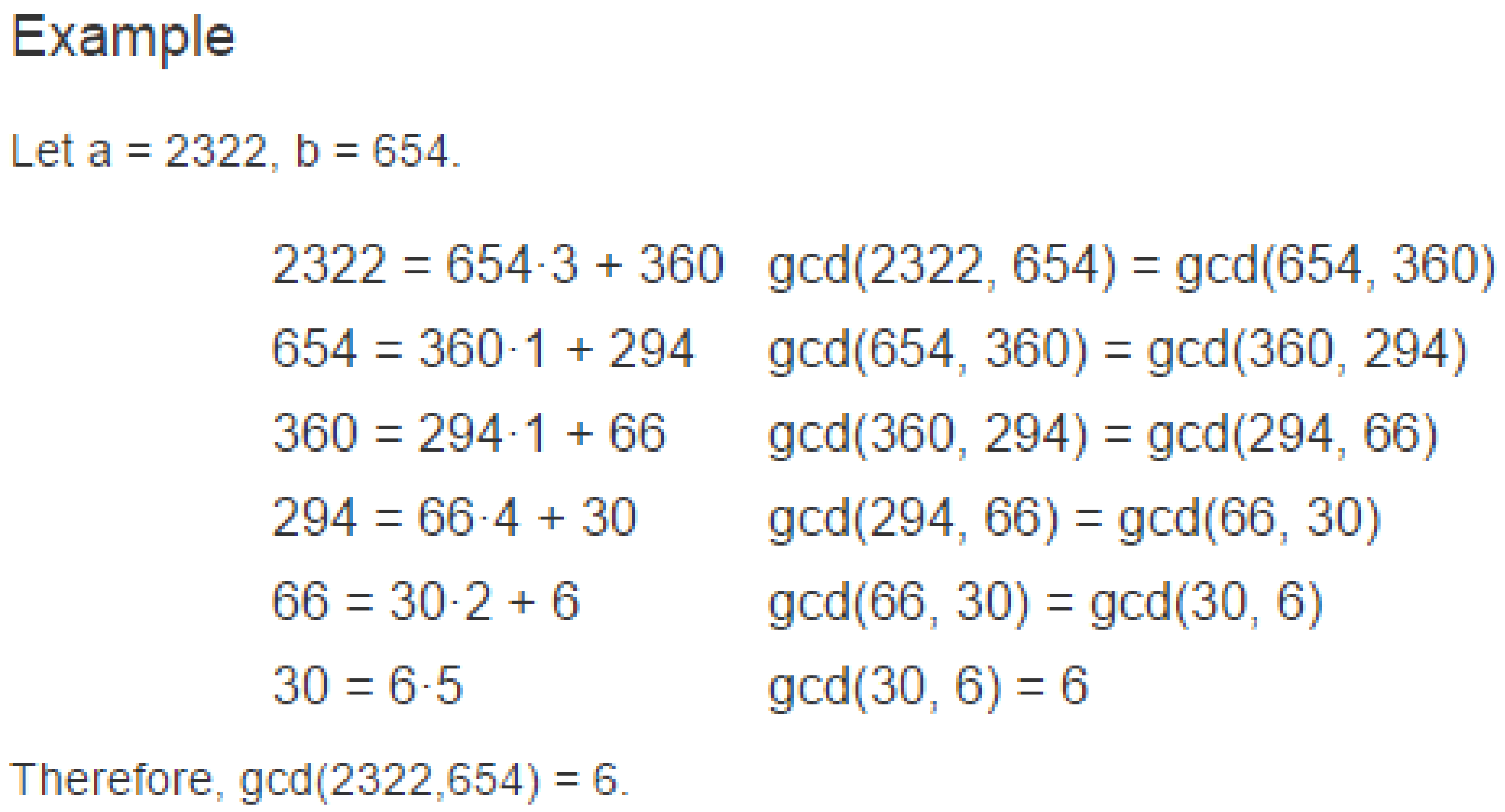
Euclidean Algorithm

a = c * f; b = d * f; a = b * t + r; r must be f’s multiple. If f is a common factor of a and b.

- The algorithm is based on the following two observations:
  1. If b|a then *gcd*(a, b) = b.
     - This is indeed so because no number (b, in particular) may have a divisor greater than the number itself (I am talking here of non-negative integers.)
  2. If a = bt + r, for integers t and r, then gcd(a, b) = gcd(b, r).

- Indeed, every common divisor of a and b also divides r. Thus gcd(a, b) divides r. But, of course, gcd(a, b)|b. Therefore, gcd(a, b) is a common divisor of b and r and hence gcd(a, b) ≤ gcd(b, r). The reverse is also true because every divisor of b and r also divides a.

Example



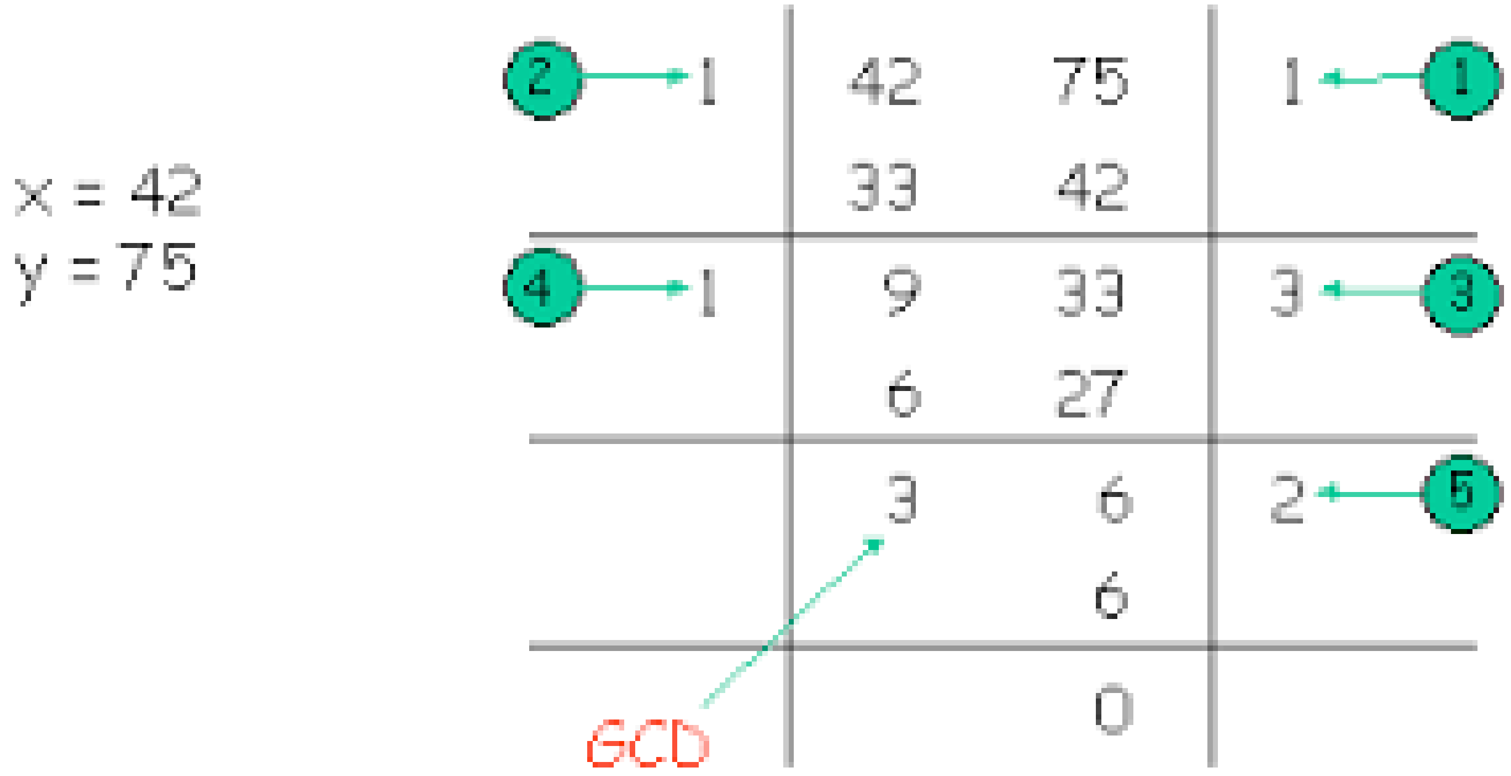
Example

Let a = 2322, b = 654.

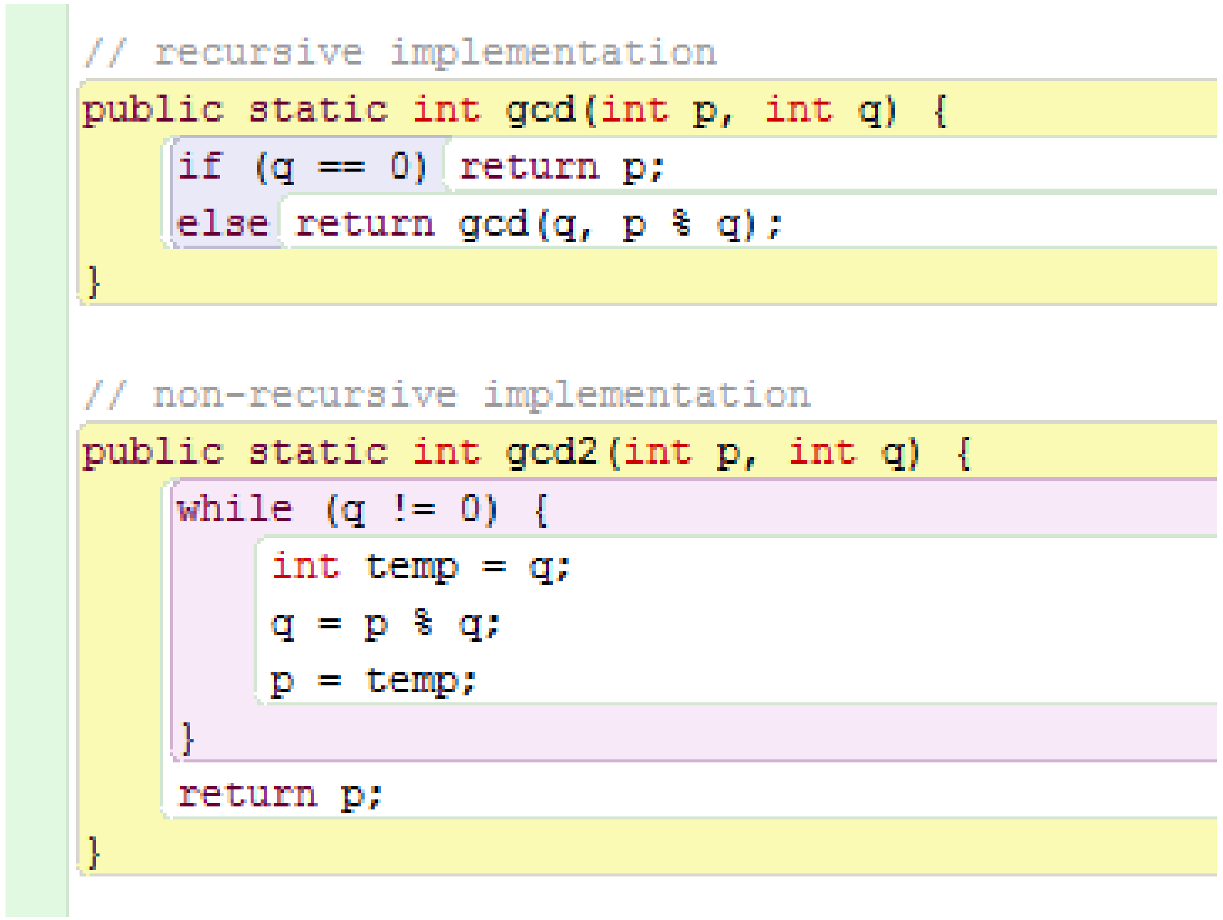
$2322 = 654 \cdot 3 + 360$	$\gcd(2322, 654) = \gcd(654, 360)$
$654 = 360 \cdot 1 + 294$	$\gcd(654, 360) = \gcd(360, 294)$
$360 = 294 \cdot 1 + 66$	$\gcd(360, 294) = \gcd(294, 66)$
$294 = 66 \cdot 4 + 30$	$\gcd(294, 66) = \gcd(66, 30)$
$66 = 30 \cdot 2 + 6$	$\gcd(66, 30) = \gcd(30, 6)$
$30 = 6 \cdot 5$	$\gcd(30, 6) = 6$

Therefore, gcd(2322,654) = 6.

Euclidean Algorithm for GCD (GCF)



Euclid’s Algorithm for GCD/GCF



```
// recursive implementation
public static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q, p % q);
}

// non-recursive implementation
public static int gcd2(int p, int q) {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}
```

Demonstration Program

EUCLID.JAVA

Demo Program:

Euclid.java

Moral of the story: good algorithm may not come from learning algorithms but from inventing them.

Finding Prime Numbers

Lecture 2

Prime Number

- A **prime number** (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself.
- A natural number greater than 1 that is not a prime number is called a composite number.

boolean isPrime(int n);

check if the number n is a prime number

Prime Number check of different efficiency:

(1) A number is prime if all of the number smaller than or equal to it can not divide it.

(2) A number is prime if all of the number smaller than or equal to half of it can not divide it.

(3) A number is prime if all of the numbers smaller than or equal to the square root of it can not divide it.

(4) A number is prime if all of the prime numbers smaller than it can not divide it.

Demonstration Program

PRIMENUMBER.JAVA

Demo Program: PrimeNumber.java

Moral of Story: Don’t get stuck at the low performance algorithms. There might be some better ways of doing things. Find the best algorithm before you perform coding.

Linear Search

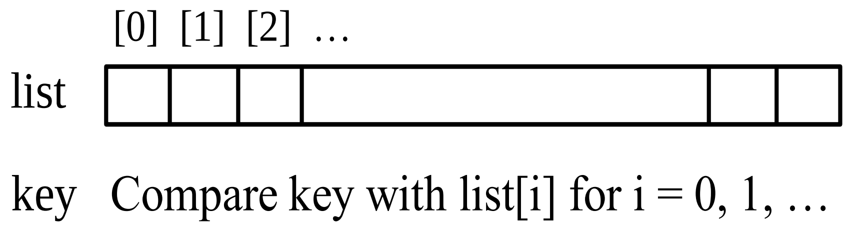
(Enhanced with Implementation from Chapter 7)

Lecture 3

Searching Arrays

Searching is the process of looking for a specific element in an array; for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. There are many algorithms and data structures devoted to searching. In this section, two commonly used approaches are discussed, ***linear search*** and ***binary search***.

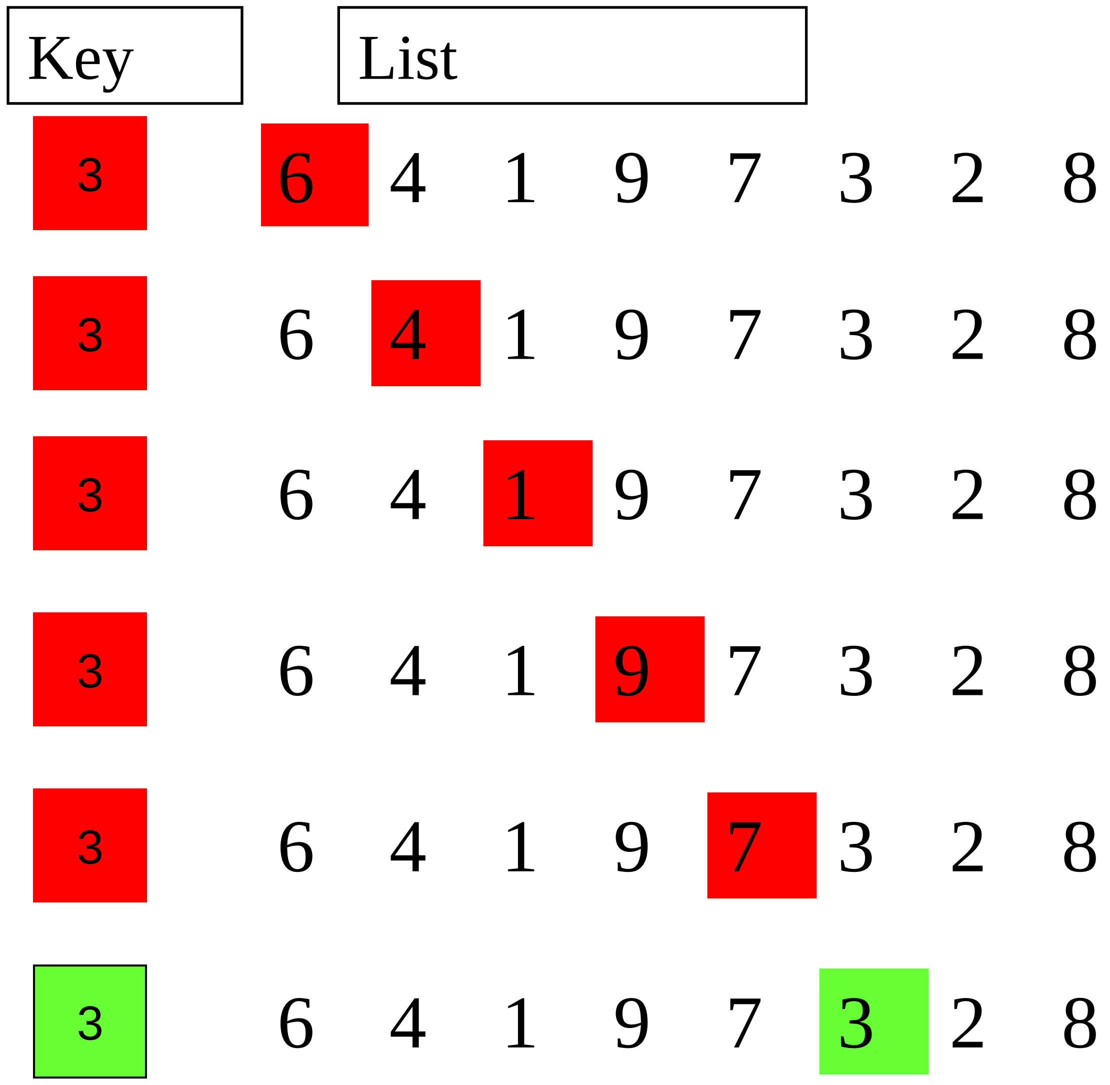
```
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++)
      if (key == list[i])
        return i;
    return -1;
  }
```



Linear Search

The linear search approach compares the key element, **<u>key</u>**, *sequentially* with each element in the array **<u>list</u>**. The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns **<u>-1</u>**.

Linear Search Animation



From Idea to Solution

```
/** The method for finding a key in the list */
public static int linearSearch(int[] list, int key) {
  for (int i = 0; i < list.length; i++)
    if (key == list[i])
      return i;
  return -1;
}
```

Trace the method

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4);  // returns 1
int j = linearSearch(list, -4); // returns -1
int k = linearSearch(list, -3); // returns 5
```

Demonstration Program

LINEARSEARCH.JAVA

Binary Search

(Enhanced with Implementation from Chapter 7)

Lecture 4

Binary Search (on sorted array)

- For binary search to work, the elements in the array must already be ordered. Without loss of generality, assume that the array is in ascending order.

```
e.g., 2 4 7 10 11 45 50 59 60 66 69 70 79
```

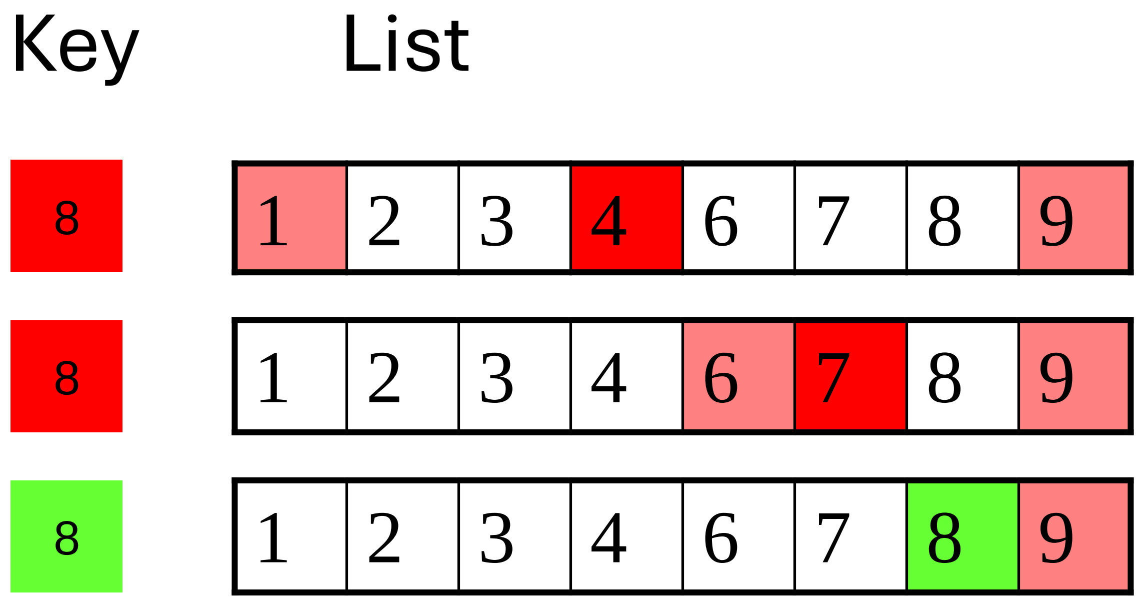
- The binary search first compares the key with the element in the middle of the array.

Binary Search, cont.

Consider the following three cases:

- If the **key** is less than the middle element, you only need to search the key in the first half of the array.
- If the **key** is equal to the middle element, the search ends with a match.
- If the **key** is greater than the middle element, you only need to search the key in the second half of the array.

Binary Search



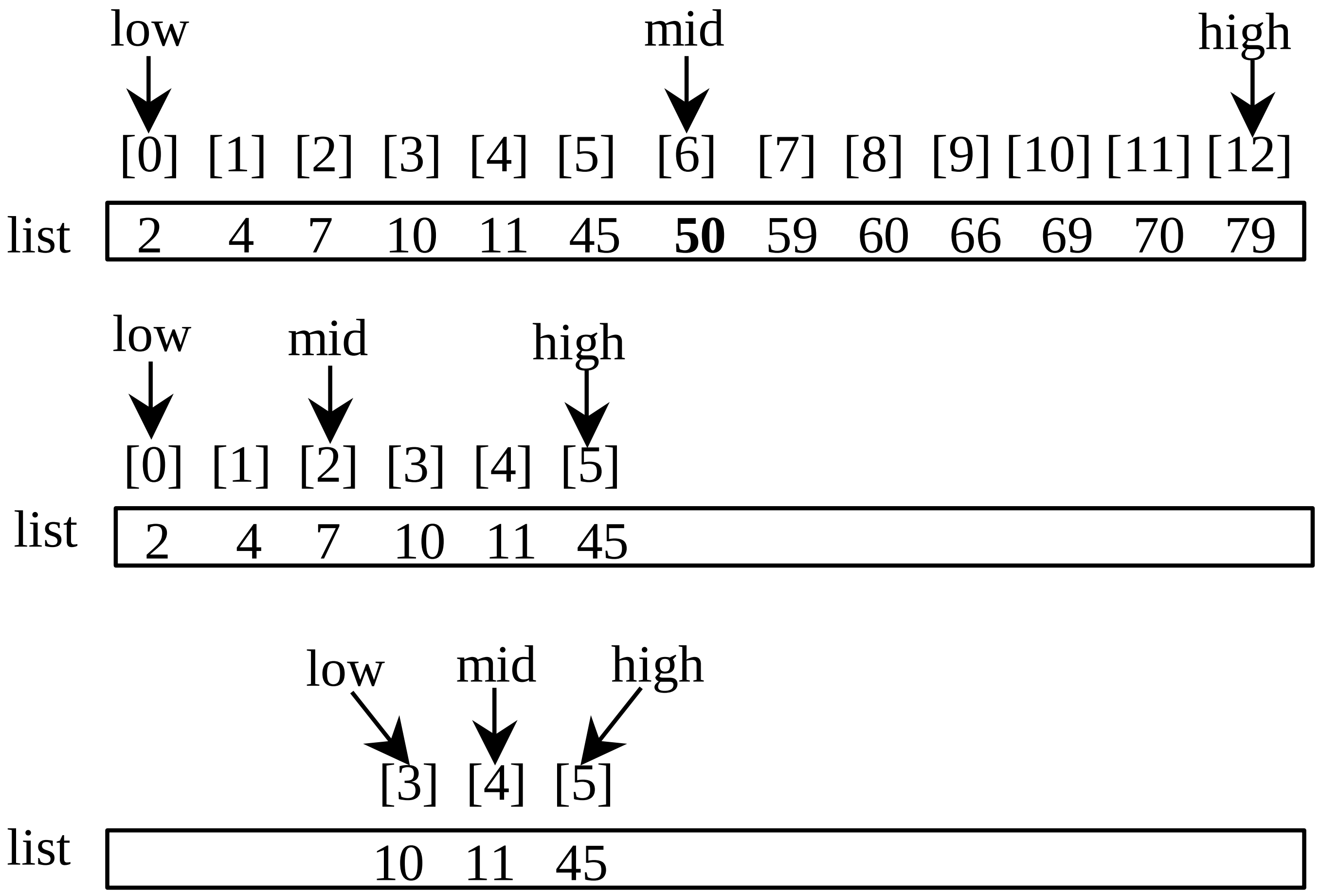
Binary Search, cont.

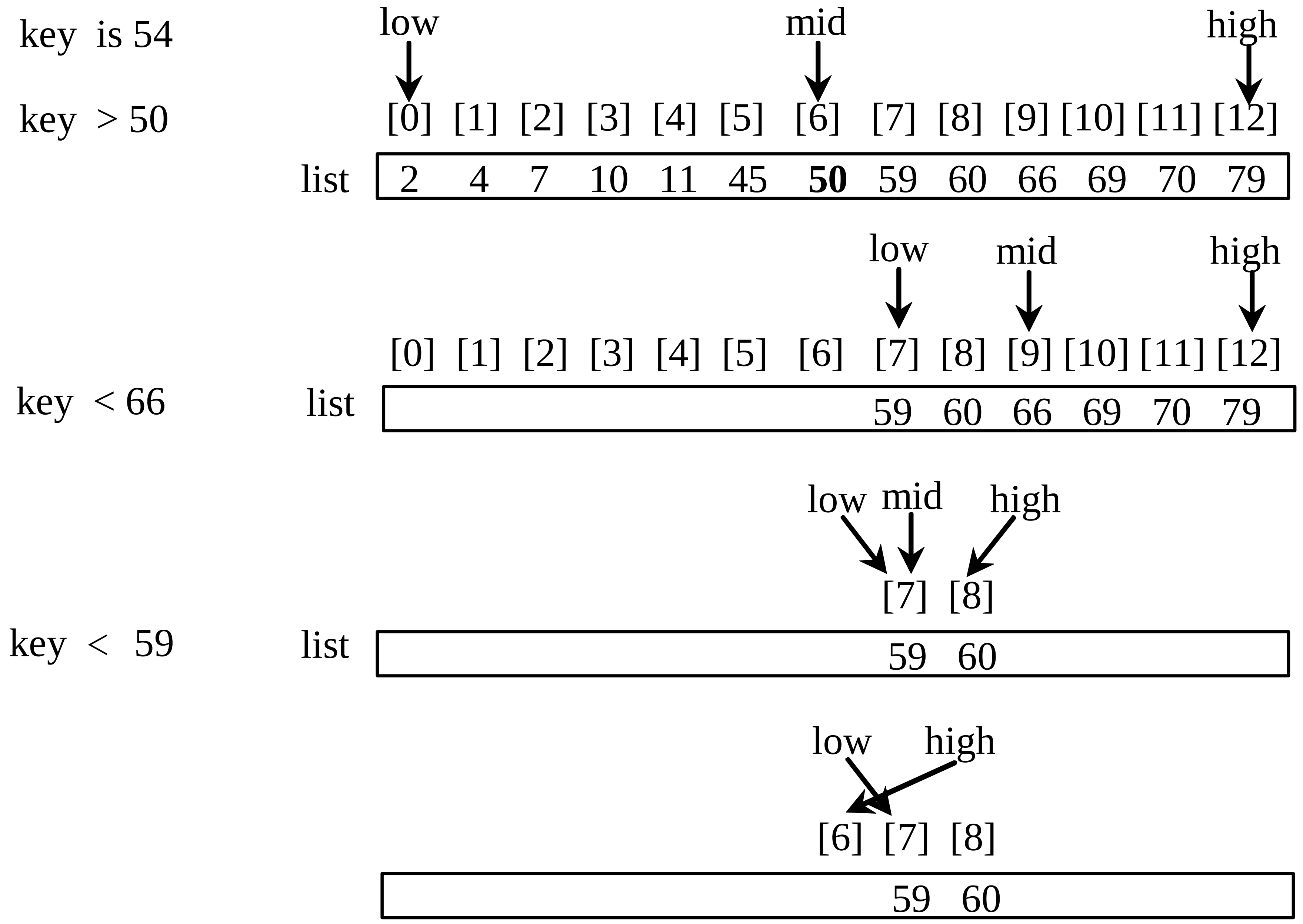
key is 11

key < 50

key > 7

key == 11




key is 54
key > 50
low
mid
high
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]
list 2 4 7 10 11 45 **50** 59 60 66 69 70 79
low
mid
high
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]
key < 66
list 59 60 66 69 70 79
low
mid
high
[7] [8]
key < 59
list 59 60
low
high
[6] [7] [8]
59 60

Binary Search, cont.

- The binarySearch method returns the index of the element in the list that matches the search key if it is contained in the list. Otherwise, it returns

```
-insertion point - 1.
  insertion point = -(return+1)
```

- The insertion point is the point at which the key would be inserted into the list.

Exemplary Binary Search Method

```
public static int binarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }
  return -1 - low;
}
```

Logarithm: Analyzing Binary Search

- BinarySearch.java, searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by c. Let T(n) denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and k=logn. Since binary search eliminates half of the input after two comparisons,

$$T(n) = T(\frac{n}{2}) + c = T(\frac{n}{2^2}) + c + c = \ldots = T(\frac{n}{2^k}) + ck = T(1) + c\log n = 1 + c\log n$$

Logarithmic Time

- Ignoring constants and smaller terms, the complexity of the binary search algorithm is *O(logn)*. An algorithm with the time complexity is called a *logarithmic algorithm O(logn)*.
- The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.

Demonstration Program

BINARYSEARCH.JAVA

Objectives

- Array.sort()
- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge Sort
- Quick Sort

Arrays.sort

Lecture 5

The Arrays.sort Method

- Since sorting is frequently used in programming, Java provides several overloaded sort methods for sorting an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers);

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars);
```

If AP test want you to write a sorting program, DO NOT USE THIS !!!
If they allow you to use it, that’s fine.

Array Class
java.util.Arrays class (sorting)

- The java.util.Arrays class contains various static methods for sorting arrays, searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types.
- `double[] numbers = {6.0, 4.4, 1.9, 3.4, 3.5} ;`
- `java.util.Arrays.sort(numbers);`

  `java.util.Arrays.parallelsort(numbers);`
- `char[] chars = {‘a’,’A’,’4’,’F’,’D’,’P’} ;`
- `java.util.Arrays.sort(chars);`

  `java.util.Arrays.parallelsort(chars);`

Sorting I: Insertion Sort

Lecture 1

Insertion Sort

```
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted
```

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step2: The sorted sublist is {2, 9}. Insert 5 into the sublist.

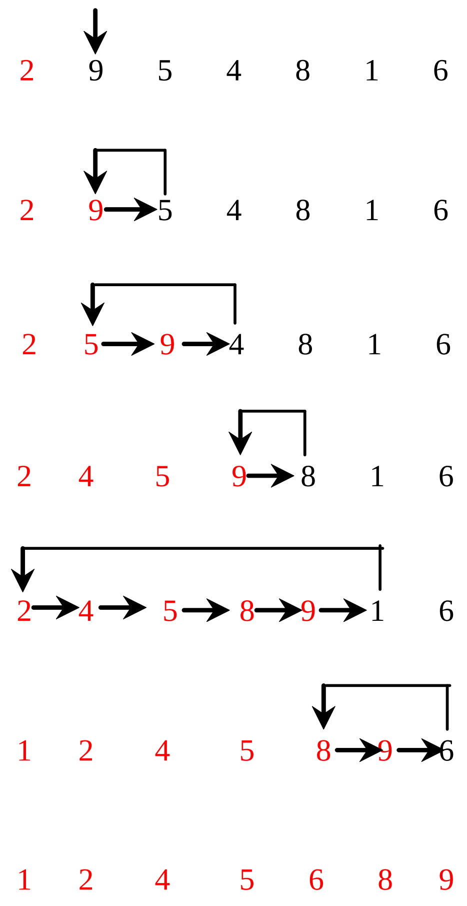
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.

Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.

Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.

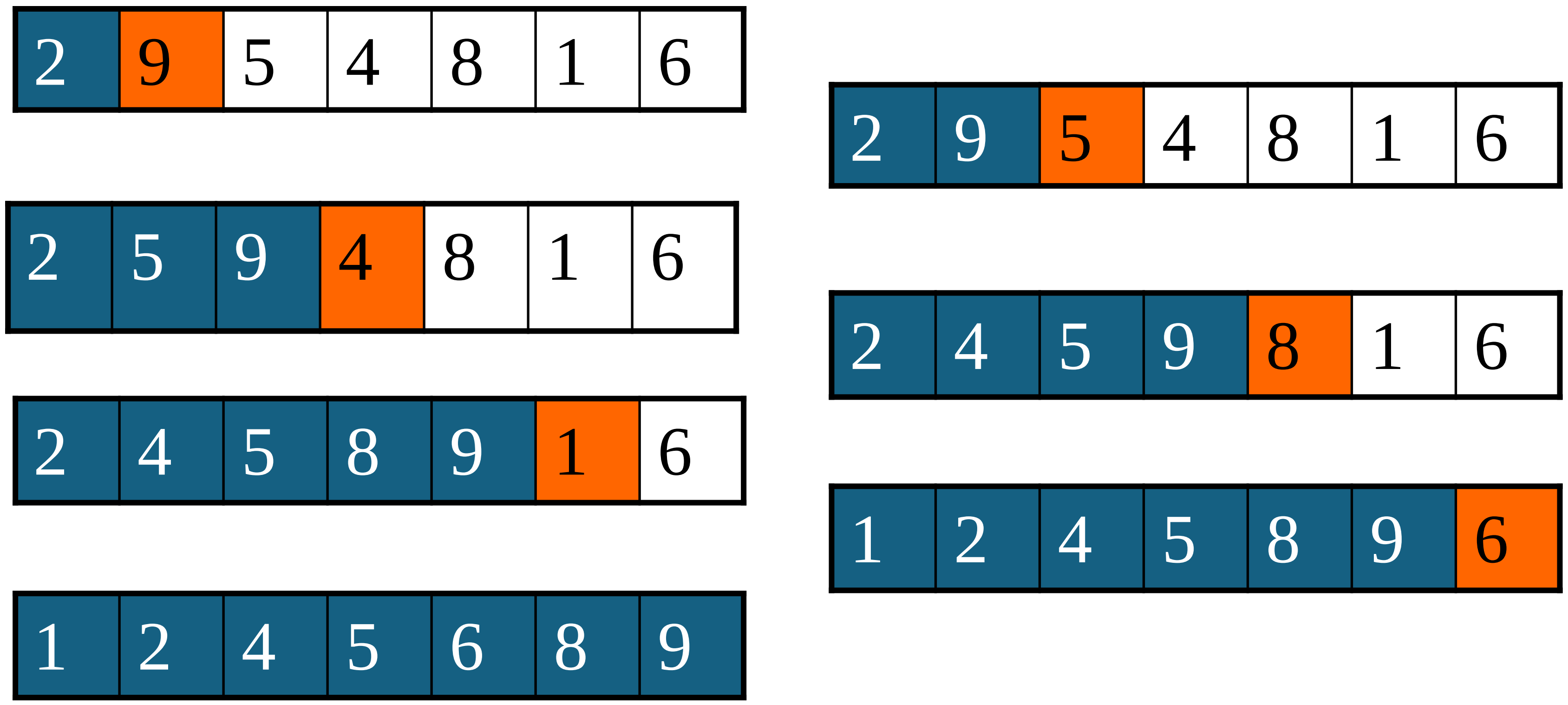
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.



Insertion Sort

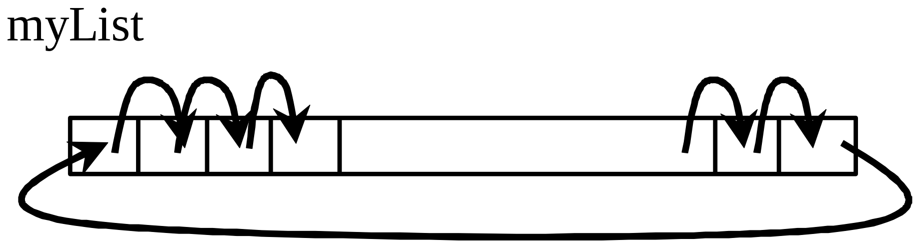
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted



Shifting Elements (Right Shifting)

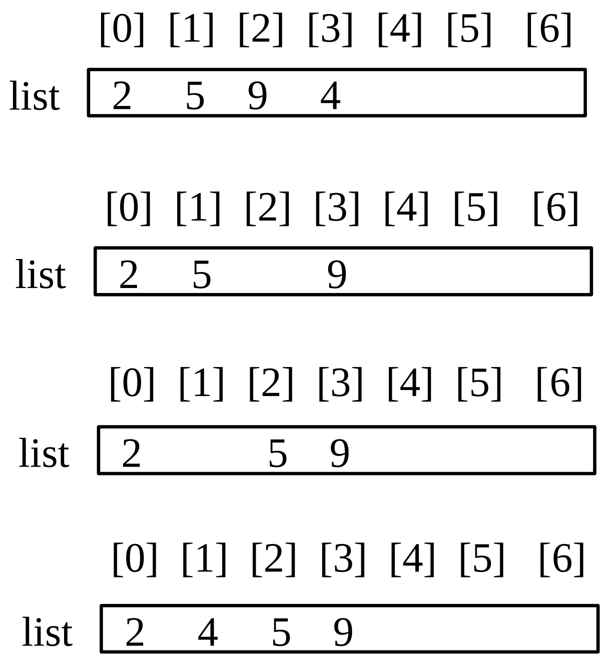
Borrowed from APCSA Chapter 7: Array Processing I

```
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;
```



How to Insert?

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.



Step 1: Save 4 to a temporary variable currentElement

Step 2: Move list[2] to list[3]

Step 3: Move list[1] to list[2]

Step 4: Assign currentElement to list[1]

From Idea to Solution

```
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
}
         list[0]

         list[0] list[1]

         list[0] list[1] list[2]

         list[0] list[1] list[2] list[3]

         list[0] list[1] list[2] list[3] ...
```

From Idea to Solution

```
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
}
```

Expand

```
double 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;
```

Analyzing Insertion Sort

- The insertion sort algorithm presented in **InsertionSort.java**, sorts a list of values by repeatedly inserting a new element into a sorted partial array until the whole array is sorted. At the ***k****th* iteration, to insert an element to a array of size ***k***, it may take ***k*** comparisons to find the insertion position, and ***k*** moves to insert the element. Let ***T(n)*** denote the complexity for insertion sort and *c* denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = 2 + c + 2 \times 2 + c \ldots + 2 \times (n - 1) + c = n^2 - n + cn$$

- Ignoring constants and smaller terms, the complexity of the selection sort algorithm is O(***n^2***).

Demonstration Program

INSERTIONSORT.JAVA

Sorting II: Selection Sort

Lecture 6

Sorting Arrays

- Sorting, like searching, is also a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces two simple, intuitive sorting algorithms: *selection sort* and *insertion sort*.

Selection Sort

- Selection sort finds the smallest number in the list and places it first.
- It then finds the smallest number remaining and places it second, and so on until the list contains only a single number.

Select 1 (the smallest) and swap it with 2 (the first) in the list

Select 2 (the smallest) and swap it with 9 (the first) in the remaining list

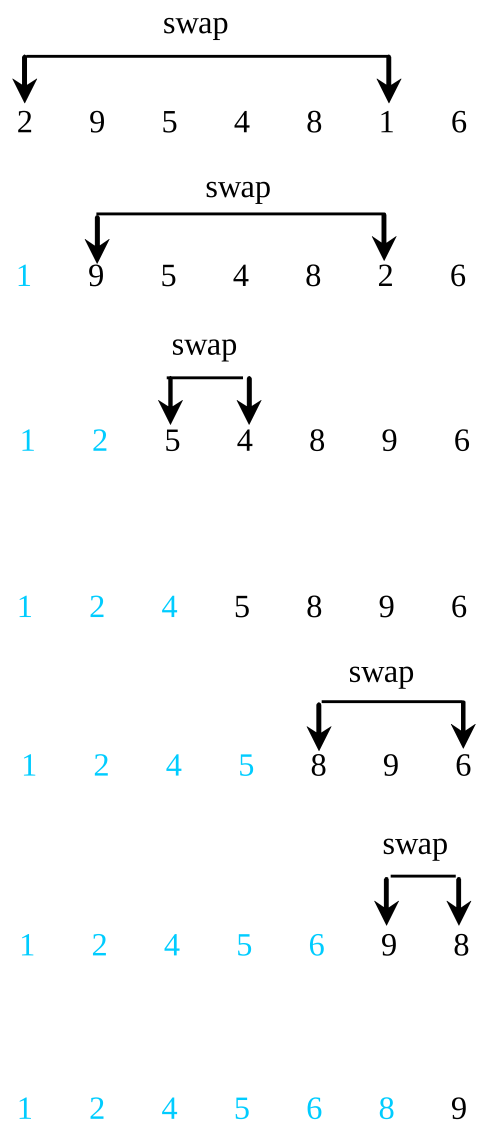
Select 4 (the smallest) and swap it with 5 (the first) in the remaining list

5 is the smallest and in the right position. No swap is necessary

Select 6 (the smallest) and swap it with 8 (the first) in the remaining list

Select 8 (the smallest) and swap it with 9 (the first) in the remaining list

Since there is only one element remaining in the list, sort is completed



The number 1 is now in the correct position and thus no longer needs to be considered.

The number 2 is now in the correct position and thus no longer needs to be considered.

The number 6 is now in the correct position and thus no longer needs to be considered.

The number 5 is now in the correct position and thus no longer needs to be considered.

The number 6 is now in the correct position and thus no longer needs to be considered.

The number 8 is now in the correct position and thus no longer needs to be considered.

From Idea to Solution

```
for (int i = 0; i < list.length; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

From Idea to Solution

```
list[0] list[1] list[2] list[3] ...               list[10]
list[0] list[1] list[2] list[3] ...               list[10]
list[0] list[1] list[2] list[3] ...               list[10]
list[0] list[1] list[2] list[3] ...               list[10]
list[0] list[1] list[2] list[3] ...               list[10]
                                ...
list[0] list[1] list[2] list[3] ...               list[10]
```

```
for (int i = 0; i < listSize; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Expand

```
double currentMin = list[i];
[redacted]
for (int j = i+1; j < list.length; j++) {
  if (currentMin > list[j]) {
    currentMin = list[j];
    [redacted]
  }
}
```

```
for (int i = 0; i < listSize; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Expand

```
double currentMin = list[i];
int currentMinIndex = i;
for (int j = i; j < list.length; j++) {
  if (currentMin > list[j]) {
    currentMin = list[j];
    currentMinIndex = j;
  }
}
```

```
for (int i = 0; i < listSize; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Expand

```
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
```

Wrap it in a Method

```
/** The method for sorting the numbers */
public static void selectionSort(double[] list) {
  for (int i = 0; i < list.length; 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;
    }
  }
}
```

Analyzing Selection Sort

- SelectionSort.java, finds the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number. The number of comparisons is *n-1* for the first iteration, *n-2* for the second iteration, and so on. Let *T(n)* denote the complexity for selection sort and *c* denote the total number of other operations such as assignments and additional comparis

$$T(n) = (n-1) + c + (n-2) + c ... + 2 + c + 1 + c = \frac{n^2}{2} - \frac{n}{2} + cn$$

- Ignoring constants and smaller terms, the complexity of the selection sort algorithm is $O(n^2)$.

Quadratic Time

An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm*. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with two nested loops are often quadratic.

Demonstration Program

SELECTIONSORT.JAVA

Demo Program:

SelectionSort.java

- InsertSort.java’s core is **circular shifting**.
- SelectionSort.java’s core is **finding mimum and swap**.

Sorting III: Bubble Sort

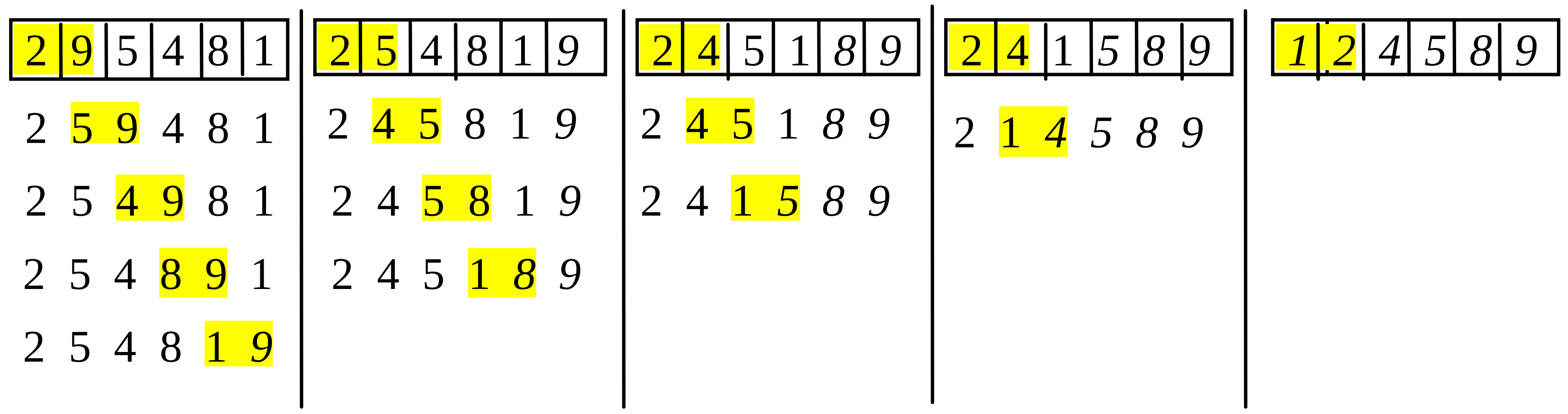
Lecture 8

Bubble Sort

**A bubble sort sorts the array in multiple phases. Each pass successively swaps the
neighboring elements if the elements are not in order.**

- The bubble sort algorithm makes several passes through the array. On each pass, **successive neighboring pairs are compared**. If a pair is in decreasing order, its values are **swapped**; otherwise, the values remain unchanged.
- The technique is called a bubble sort or sinking sort, because the smaller values gradually “bubble” their way to the top and the larger values sink to the bottom. After the first pass, the last element becomes the largest in the array. After the second pass, the second-to-last element becomes the second largest in the array. This process is continued until all elements are sorted.

Bubble Sort



(a) 1st pass (b) 2nd pass (c) 3rd pass (d) 4th pass (e) 5th pass

Bubble sort time: $O(n^2)$

$$(n-1)+(n-2)+\ldots+2+1=\frac{n^2}{2}-\frac{n}{2}$$

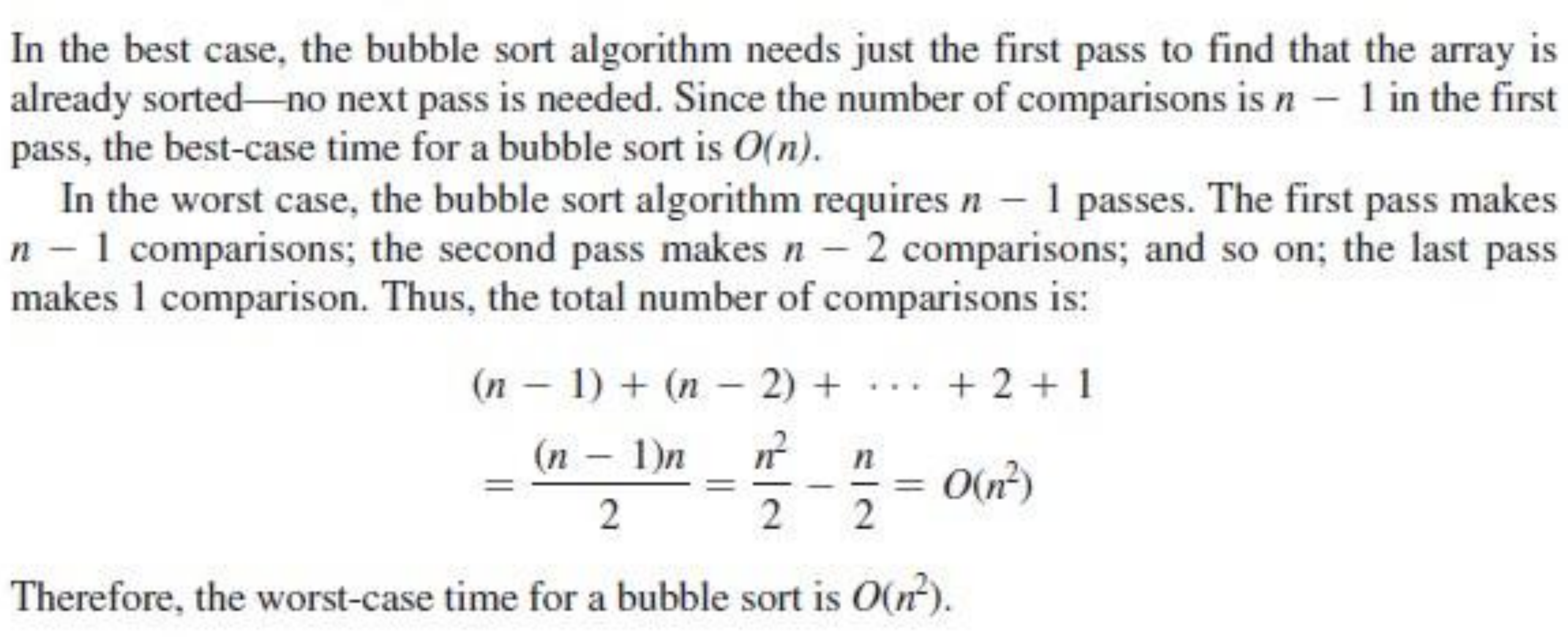
From idea to Solution

```
for (int k = 1; k<list.length; k++){
 // Perform the kth pass
 for (int i = 0; i<list.length-k; i++){
 if (list[i] > list[i + 1])
 swap list[i] with list[i + 1];
 }
}
```

Improved Bubble Sort Algorithm

```
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
 // Array may be sorted and next pass not needed
 needNextPass = false;
 // Perform the kth pass
  for (int i = 0; i < list.length - k; i++) {
      if (list[i] > list[i + 1]) {  swap list[i] with list[i + 1];
         needNextPass = true; // Next pass still needed
      }
   }
}
```

Time Complexity



In the best case, the bubble sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed. Since the number of comparisons is $n - 1$ in the first pass, the best-case time for a bubble sort is $O(n)$.

In the worst case, the bubble sort algorithm requires $n - 1$ passes. The first pass makes $n - 1$ comparisons; the second pass makes $n - 2$ comparisons; and so on; the last pass makes 1 comparison. Thus, the total number of comparisons is:

$$(n - 1) + (n - 2) + \cdots + 2 + 1$$
$$= \frac{(n - 1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Therefore, the worst-case time for a bubble sort is $O(n^2)$.

Demonstration Program

BUBBLESORT.JAVA

Demo Program:

BubbleSort.java

- BubbleSort.java's Core: **Successive Comparison and Swap**.

Sorting IV: Merge Sort

Lecture 9

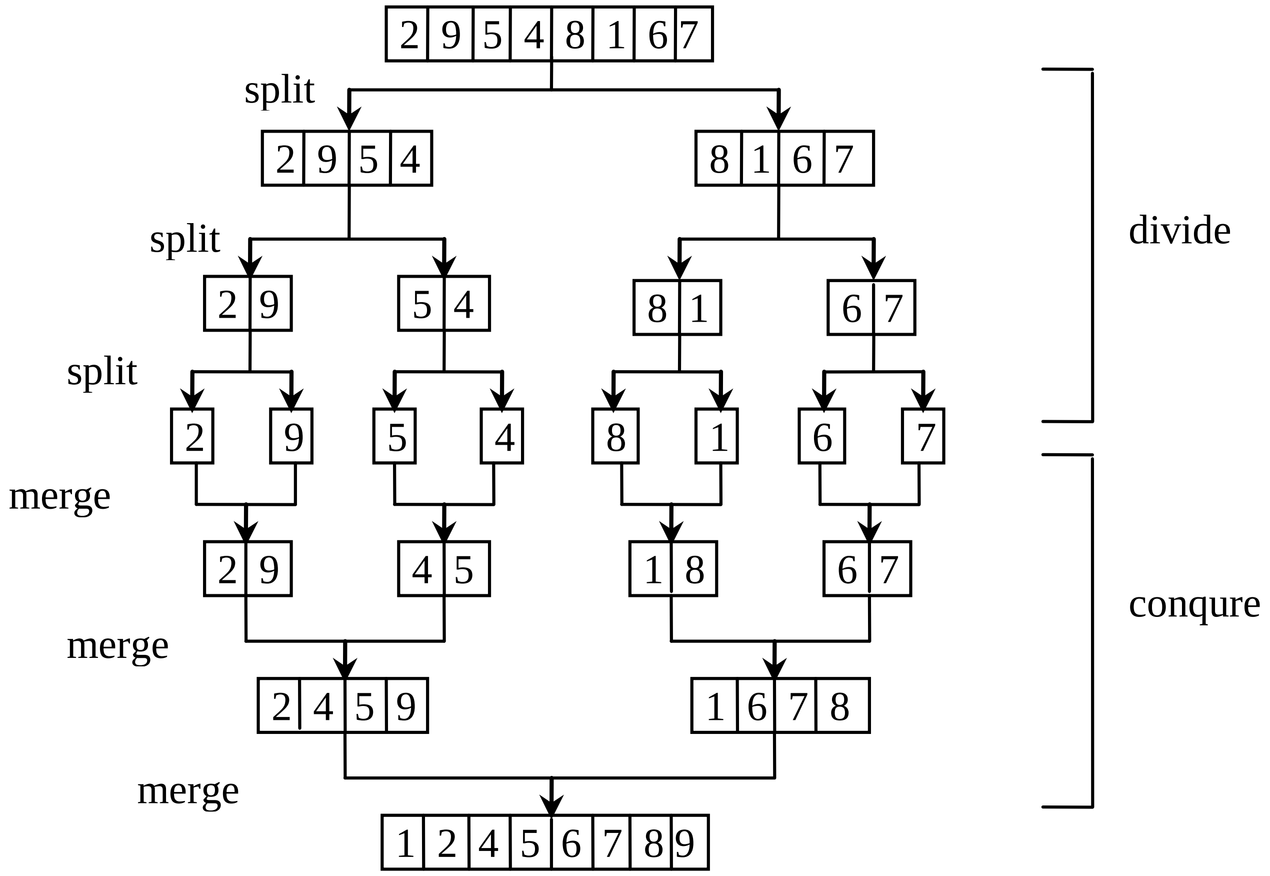
Merge Sort

The merge sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, merge them.

Merge Sort Algorithm

```
public static void mergeSort(int[] list) {
    if (list.length > 1) {
       int low = 0, high = list.length-1;
       int mid = (low+high)/2;
       mergeSort(list[0 ... mid+ 1]);
       mergeSort(list[mid + 1 ... list.length]);
       merge list[0 ... mid+ 1] with
       list[mid + 1 ... list.length];
    }
} // First half will be a little bit longer (if length is odd)
```

Merge Sort


2 9 5 4 8 1 6 7
split
2 9 5 4
8 1 6 7
split
2 9
5 4
8 1
6 7
split
2
9
5
4
8
1
6
7
merge
2 9
4 5
1 8
6 7
merge
2 4 5 9
1 6 7 8
merge
1 2 4 5 6 7 8 9
divide
conqure

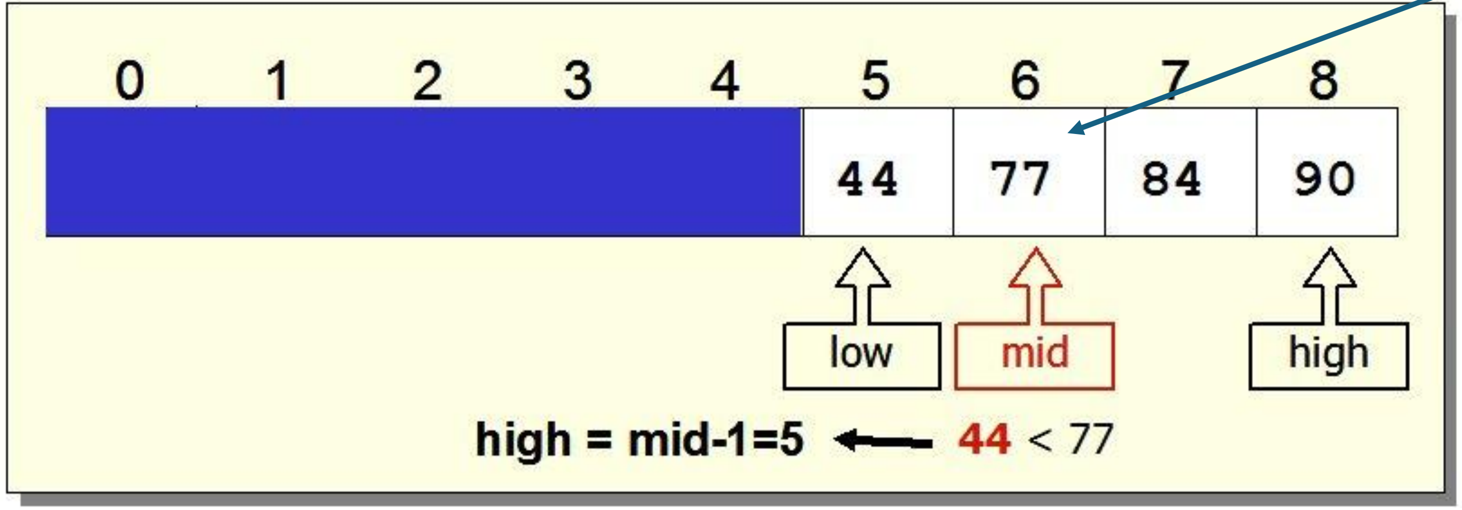
We use the same low, mid, high system as the Binary Search

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

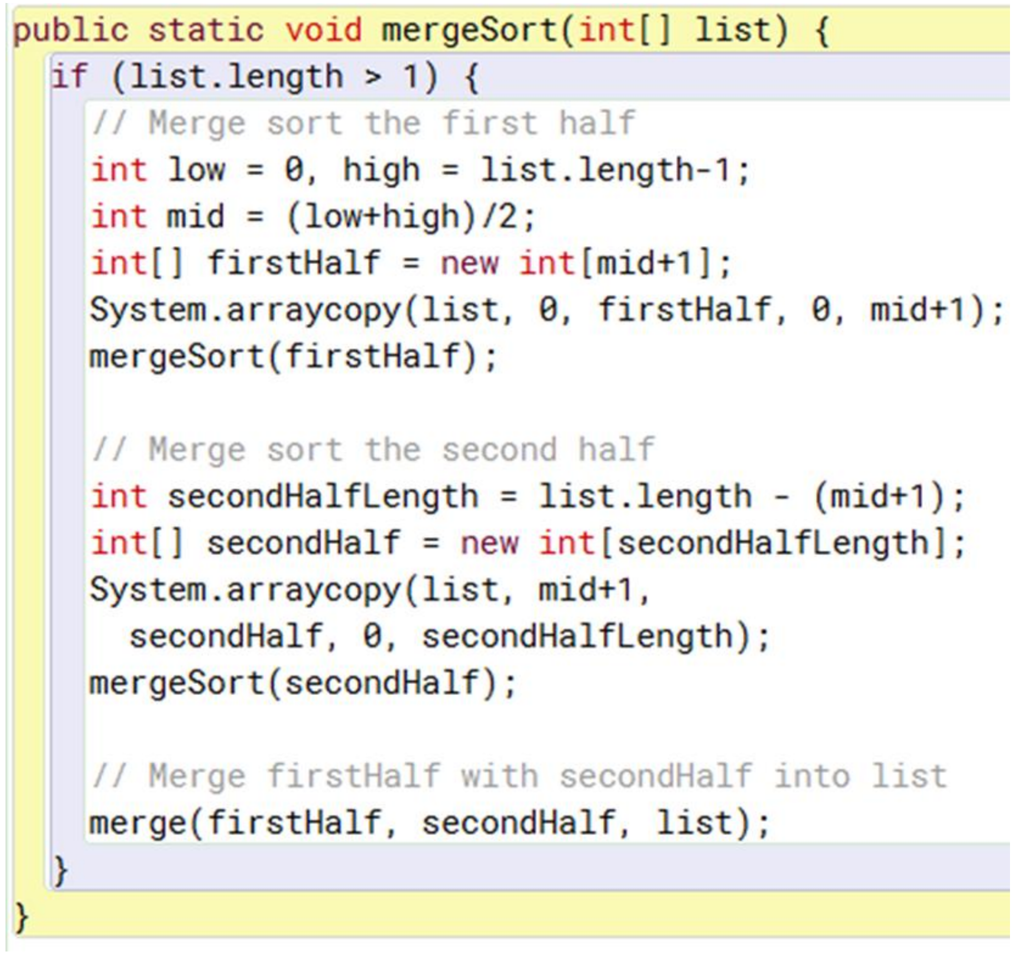
search(44)

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

mid belongs to first half

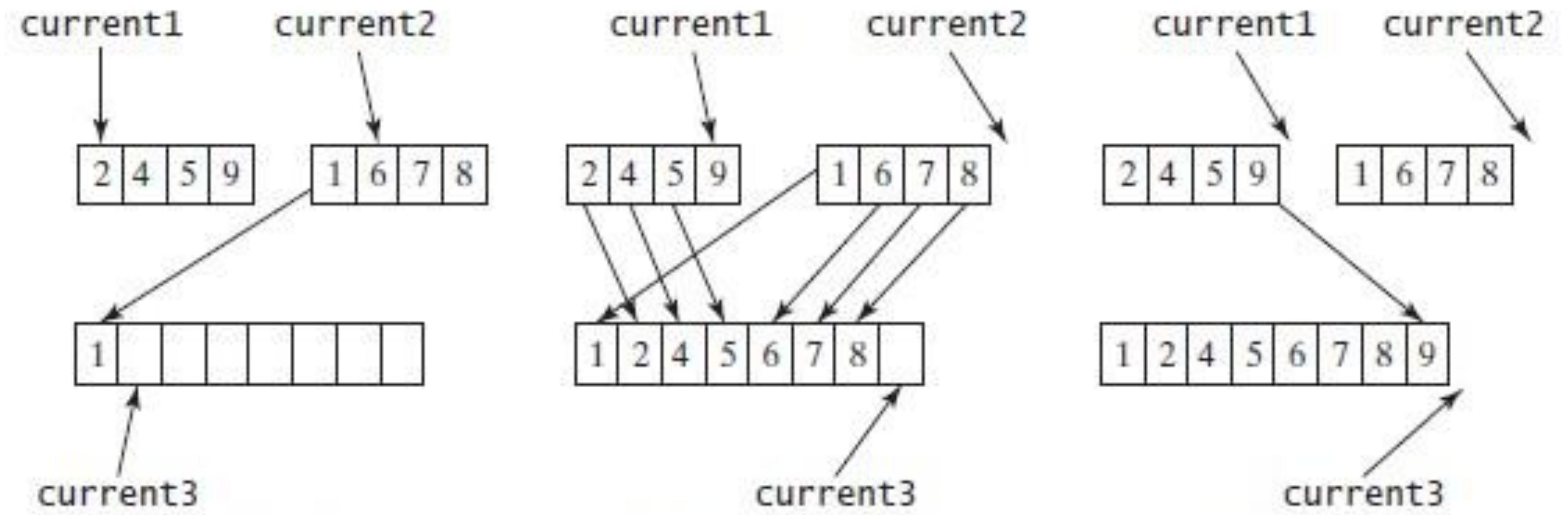


Top Level mergeSort Method



```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);
  }
}
```

Merge

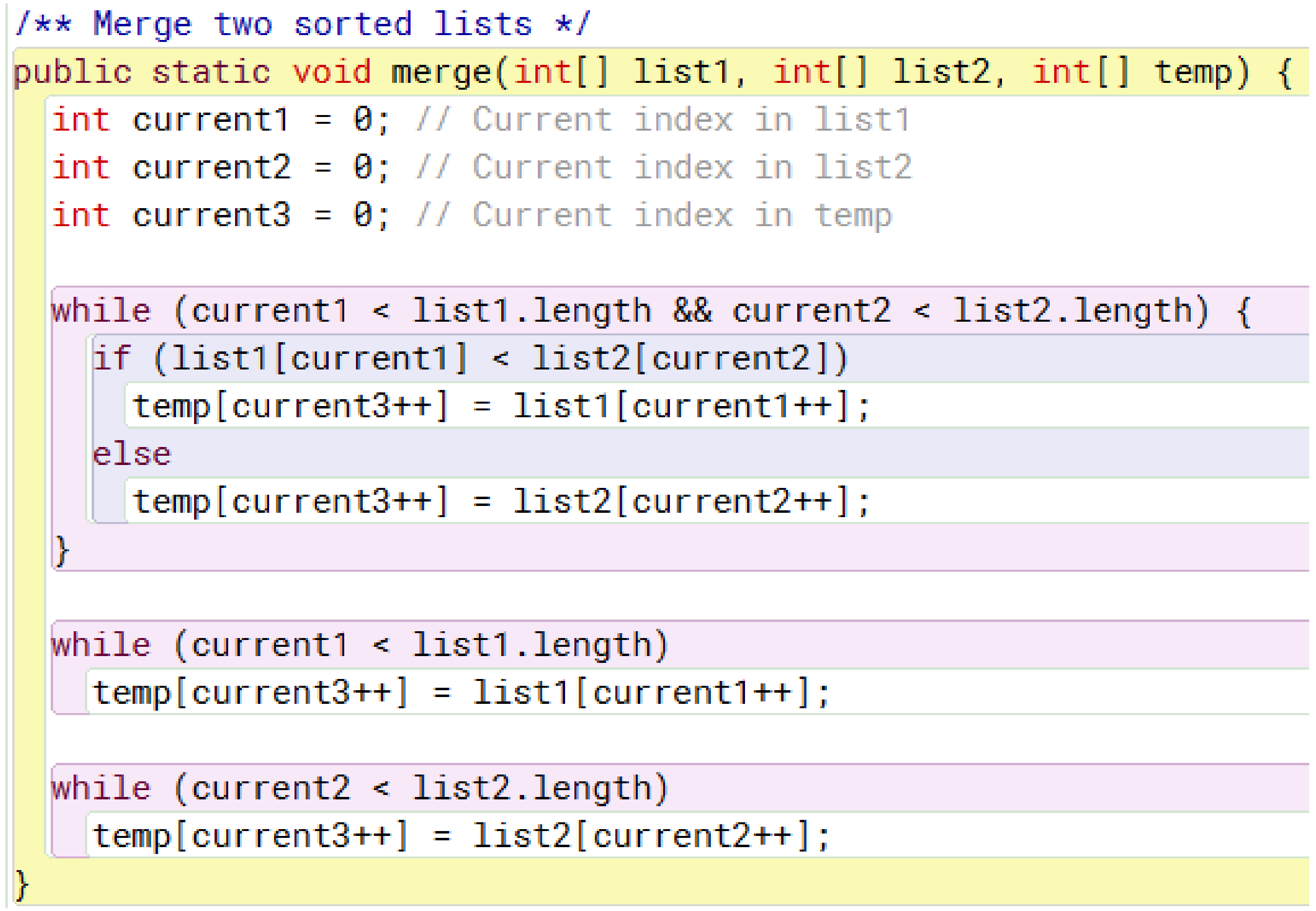


(a) After moving 1 to temp

(b) After moving all the elements in list2 to temp

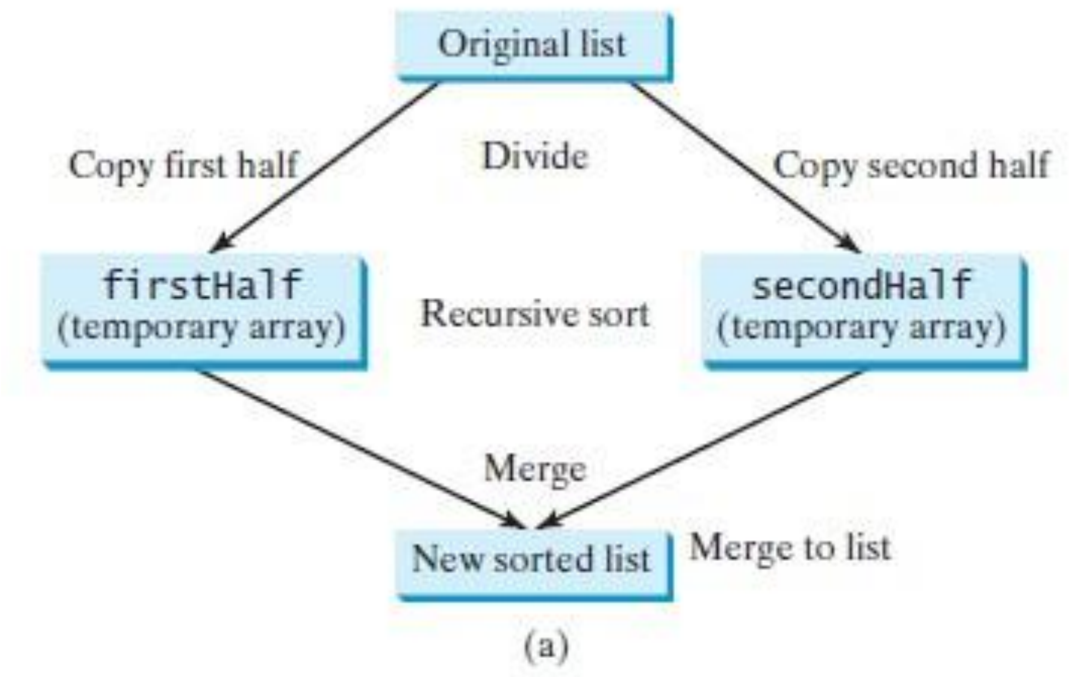
(c) After moving 9 to temp

merge Method

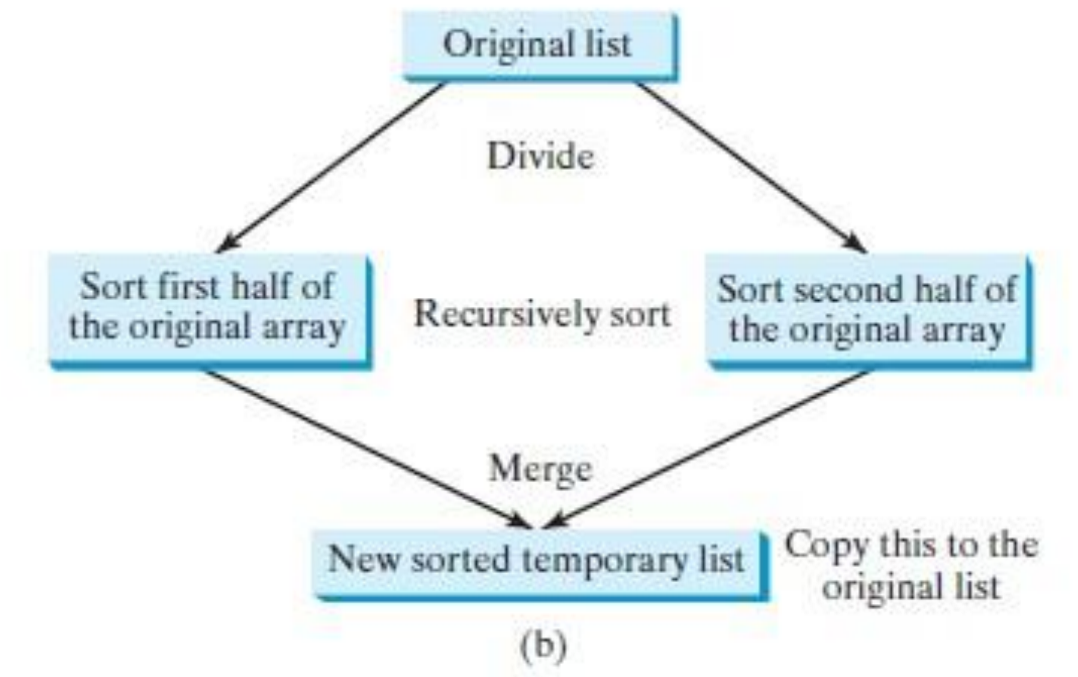


```
/** Merge two sorted lists */
public static void merge(int[] list1, int[] list2, int[] temp) {
  int current1 = 0; // Current index in list1
  int current2 = 0; // Current index in list2
  int current3 = 0; // Current index in temp

  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++];
}
```

Data Structures



(a)


Original list
Divide
Sort first half of the original array
Recursively sort
Sort second half of the original array
Merge
New sorted temporary list
Copy this to the original list

(b)

Merge Sort Time

Let T(n) denote the time required for sorting an array of n elements using merge sort. Without loss of generality, assume n is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + mergetime$$

Merge Sort Time

The first *T(n/2)* is the time for sorting the first half of the array and the second *T(n/2)* is the time for sorting the second half. To merge two subarrays, it takes at most *n-1* comparisons to compare the elements from the two subarrays and *n* moves to move elements to the temporary array. So, the total time is *2n-1*. Therefore,

$$T(n) = 2T(\frac{n}{2}) + 2n - 1 = 2(2T(\frac{n}{4}) + 2\frac{n}{2} - 1) + 2n - 1 = 2^2 T(\frac{n}{2^2}) + 2n - 2 + 2n - 1$$

$$= 2^k T(\frac{n}{2^k}) + 2n - 2^{k-1} + \ldots + 2n - 2 + 2n - 1$$

$$= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + \ldots + 2n - 2 + 2n - 1$$

$$= n + 2n \log n - 2^{\log n} + 1 = 2n \log n + 1 = O(n \log n)$$

Demonstration Program

MERGESORT.JAVA

Demo Program:

MergeSort.java

- MergeSort’s core is Divide and merge on integration.
- Arrays.sort uses merge sort algorithm. Use extra memory space to cut time.

Sorting V: Quick Sort

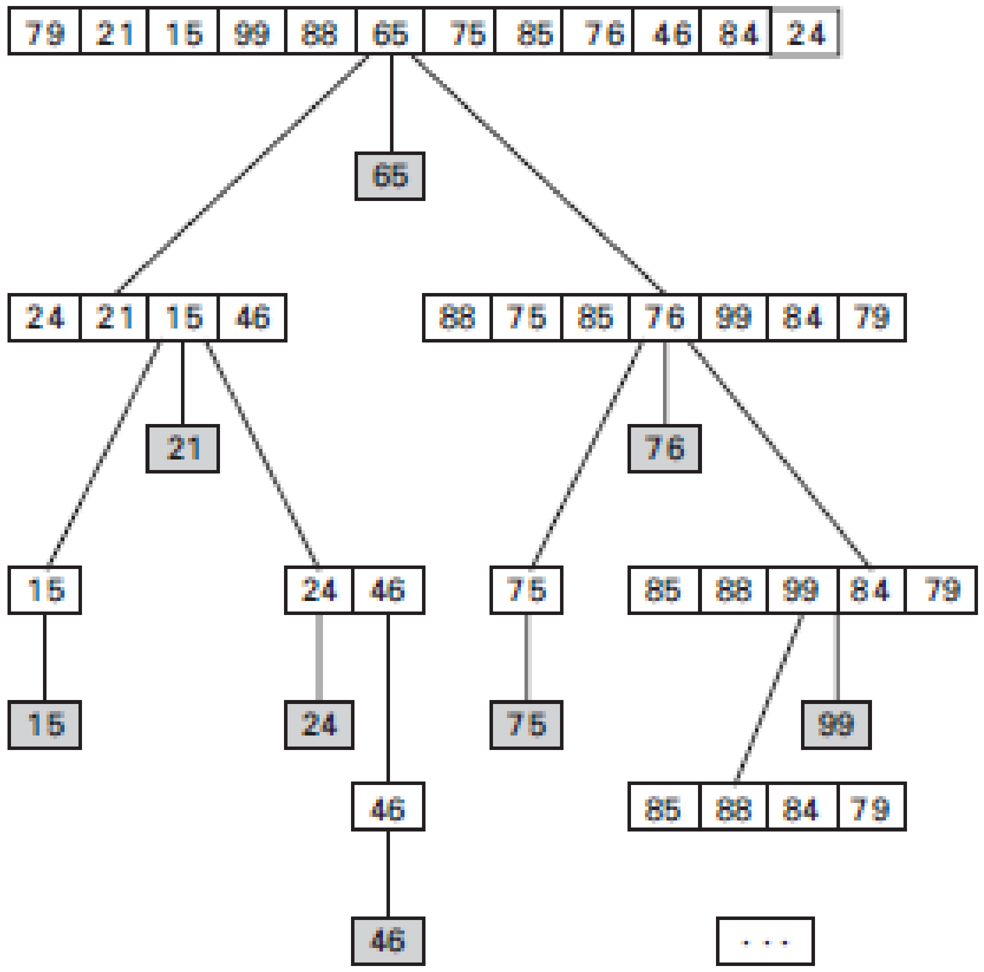
(Non-AP Topic)

Lecture 10

Quick Sort

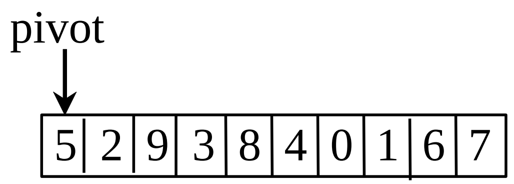
Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the **pivot**, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

Conceptual Idea of Quick Sort

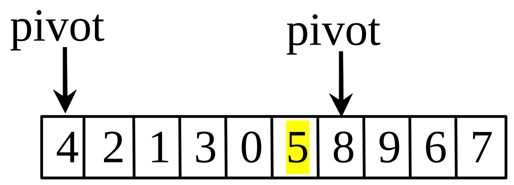


Izquierda: 24, 21, 15, 46
Pivote: 65
Derecha: 88, 75, 85, 76, 99, 84, 79

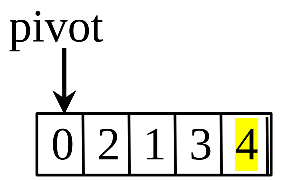
Quick Sort



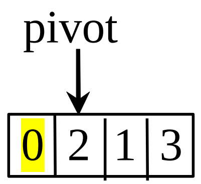
(a) The original array



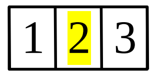
(b)The original array is partitioned



(c) The partial array (4 2 1 3 0) is partitioned

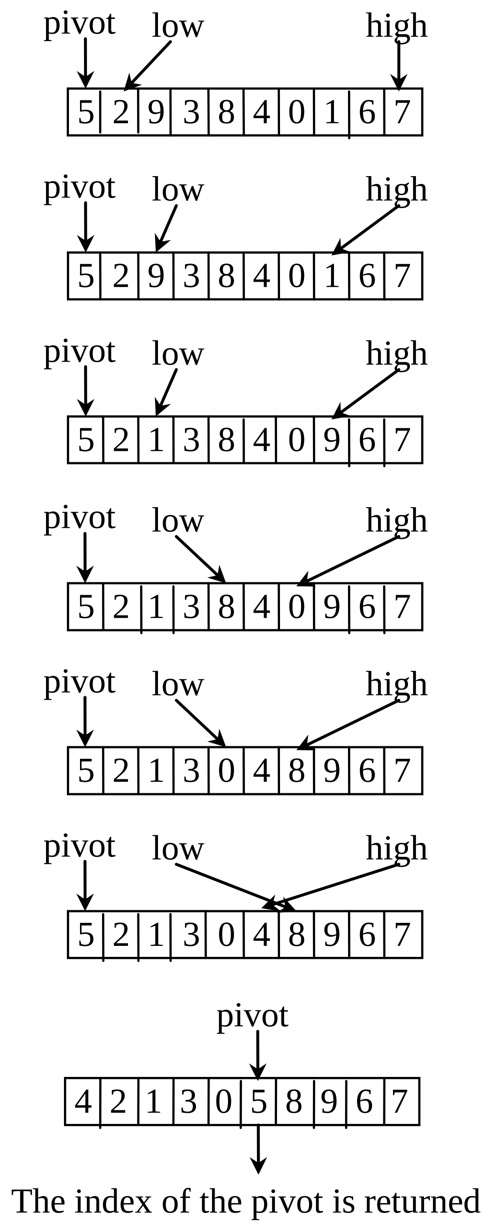


(d) The partial array (0 2 1 3) is partitioned



(e) The partial array (2 1 3) is partitioned

Partition



pivot
low
high
5 2 9 3 8 4 0 1 6 7
(a) Initialize pivot, low, and high
5 2 9 3 8 4 0 1 6 7
(b) Search forward and backward
5 2 1 3 8 4 0 9 6 7
(c) 9 is swapped with 1
5 2 1 3 8 4 0 9 6 7
(d) Continue search
5 2 1 3 0 4 8 9 6 7
(e) 8 is swapped with 0
5 2 1 3 0 4 8 9 6 7
(f) when high < low, search is over
4 2 1 3 0 5 8 9 6 7
(g) pivot is in the right place
The index of the pivot is returned

Top-down Partition and Sort

```
private static void quickSort(int[] list, int first, int last) {
   if (last > first) {  // put use the first element as pivot.
     int pivotIndex = partition(list, first, last); //relocate first
     quickSort(list, first, pivotIndex - 1); // sort first part
     quickSort(list, pivotIndex + 1, last); // sort second part
   }
 }
```

Partition Algorithm

```
private static int partition(int[] list, int first, int last) {
 (1)  int pivot = list[first]; int low = first + 1; int high = last;  //Set the pivot, low and high
 (2)   while (high > low) {
         //  Search forward from left
         while (low <= high && list[low] <= pivot) low++;  // stop when low hit a bigger one
         // Search backward from right
         while (low <= high && list[high] > pivot) high--; // stop when high hit a smaller one
         // Swap two elements in the list
         if (high > low) { int temp = list[high]; list[high] = list[low]; list[low] = temp; }
       } // when stopped, high <= low.  Anything above high is bigger than pivot
 (3) while (high > first && list[high] >= pivot) high--;  // find the right pivot location
 (4)  // Swap pivot with list[high]
         if (pivot > list[high]) { list[first] = list[high]; list[high] = pivot; return high; } // pivot it not high
         else { return first; } // pivot is first
}
```

Quick Sort Time

To partition an array of *n* elements, it takes *n* comparisons and *n* moves in the worst case. So, the time required for partition is *O(n)*.

In the worst case, each time the pivot divides the array into one big subarray with the other empty. The size of the big subarray is one less than the one before divided. The algorithm requires time:

$$(n-1)+(n-2)+\ldots+2+1=O(n^2)$$

Worst-Case Time

In the worst case, each time the pivot divides the array into one big subarray with the other empty. The size of the big subarray is one less than the one before divided. The algorithm requires:

$$(n-1)+(n-2)+\ldots+2+1=O(n^2)$$

Best-Case Time

In the best case, each time the pivot divides the array into two parts of about the same size. Let *T(n)* denote the time required for sorting an array of elements using quick sort. So,

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$$

Average-Case Time

On the average, each time the pivot will not divide the array into two parts of the same size nor one empty part. Statistically, the sizes of the two parts are very close. So the average time is *O(logn)*. The exact average-case analysis is beyond the scope of this book.

Demonstration Program

QUICKSORT.JAVA

Demo Program:

QuickSort.java

- Quick Sort is based on pick a pivot, throws bigger one to its right and smaller one to its left. **Pivot and Throw.**