Chapter 9
Searching and Sorting

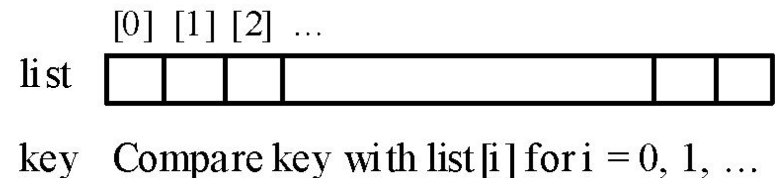
CS1: Java Programming
Colorado State University

Original slides by Daniel Liang
Modified slides by Kris Brown

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;
  }
}
```



Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Linear Search

The linear search approach compares the key element, key, *sequentially* with each element in the array list. 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 -1.

animation

Linear Search Animation

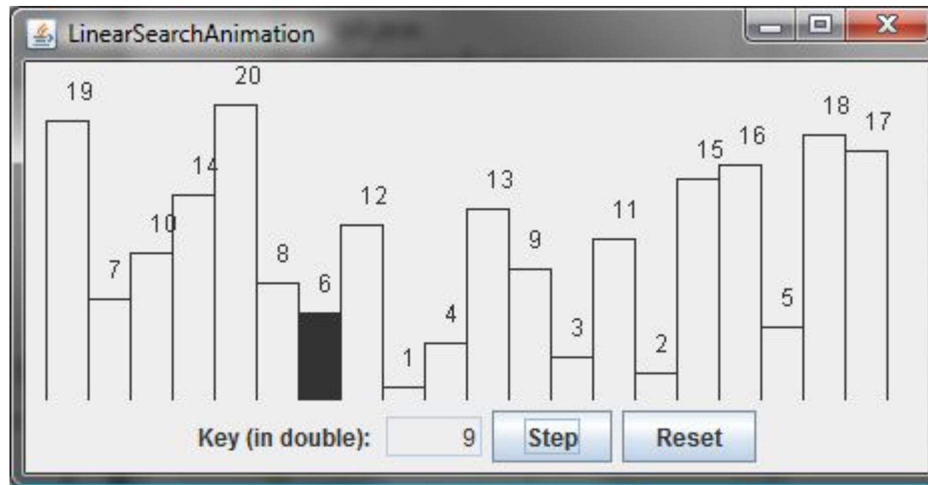
Key	List							
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

animation

Linear Search Animation

http://www.cs.armstrong.edu/liang/animation/web/LinearSearch.html



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
```

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Binary Search

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.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

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.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

animation

Binary Search

Key List

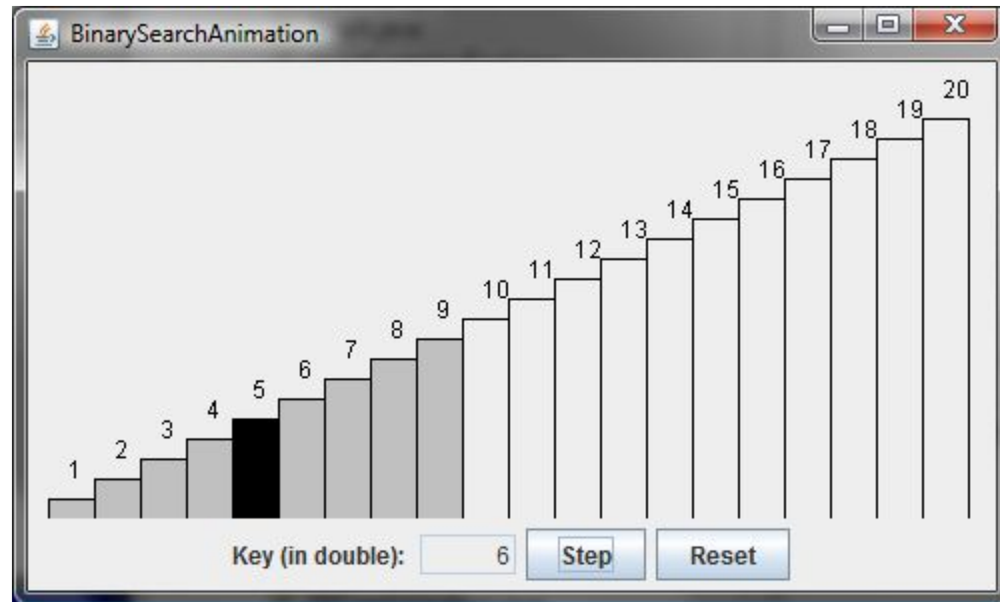
Key								
8	1	2	3	4	6	7	8	9
8	1	2	3	4	6	7	8	9
8	1	2	3	4	6	7	8	9

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

animation

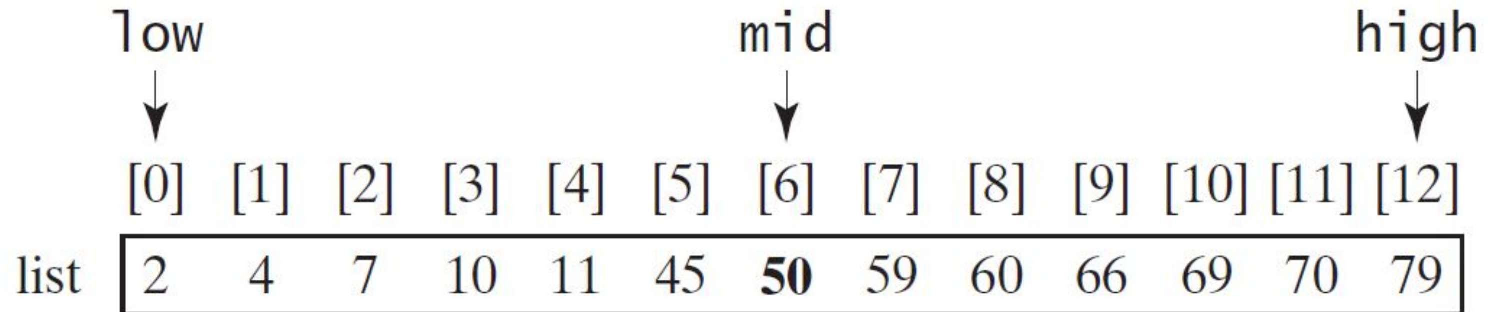
Binary Search Animation

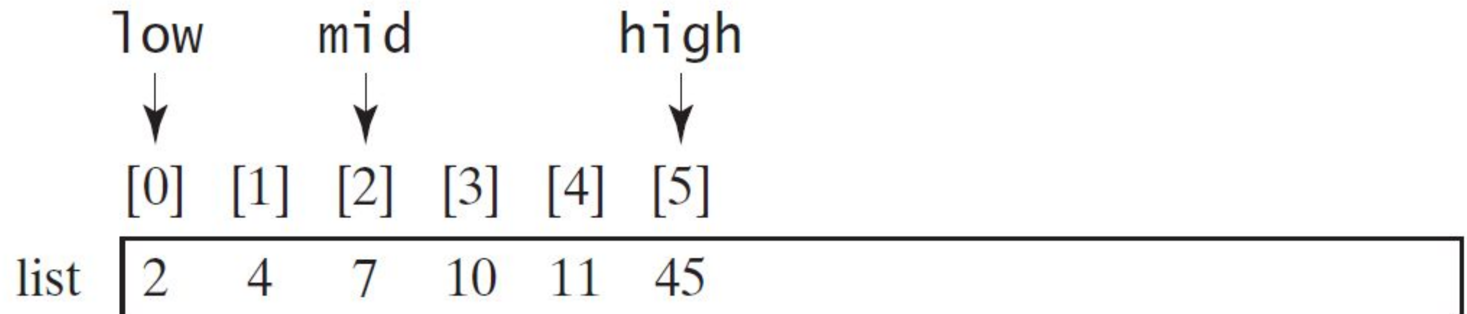
http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html

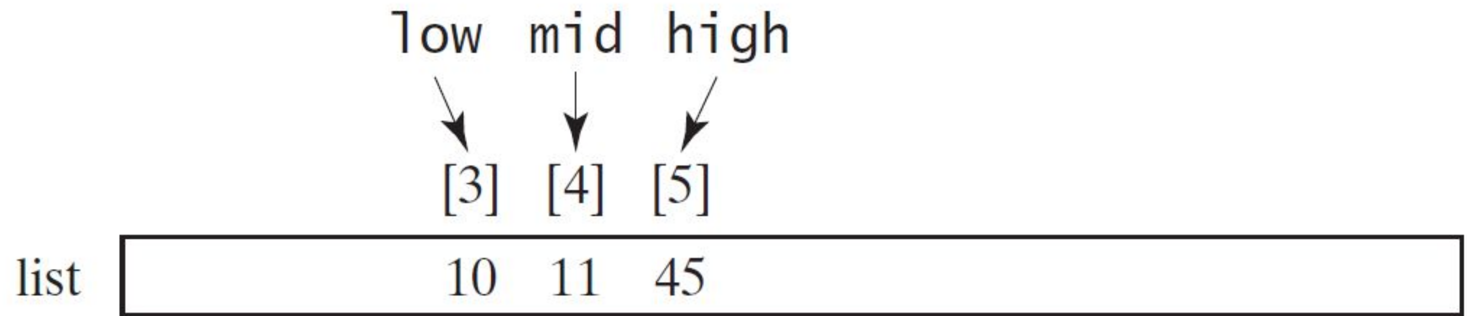


Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Binary Search, cont.







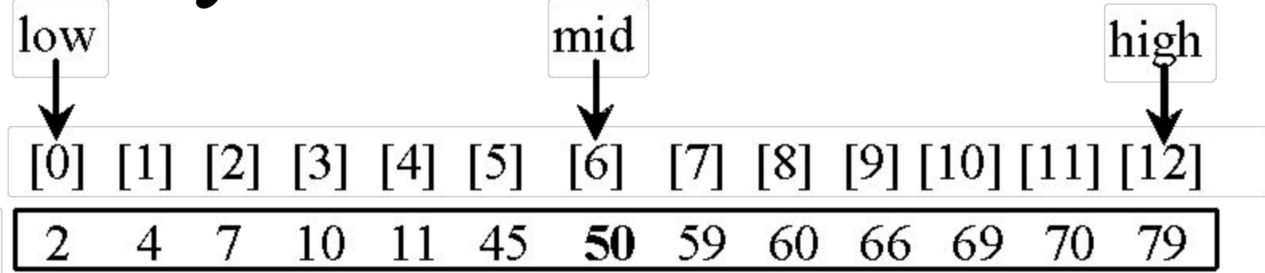
Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Binary Search, cont.

key is 54

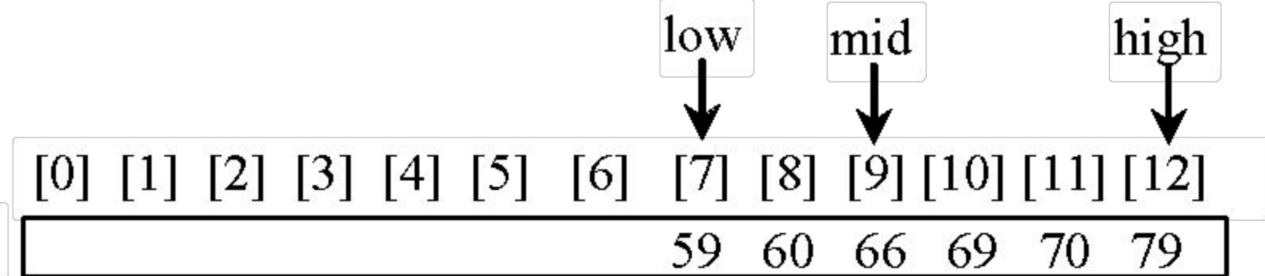
key > 50

list



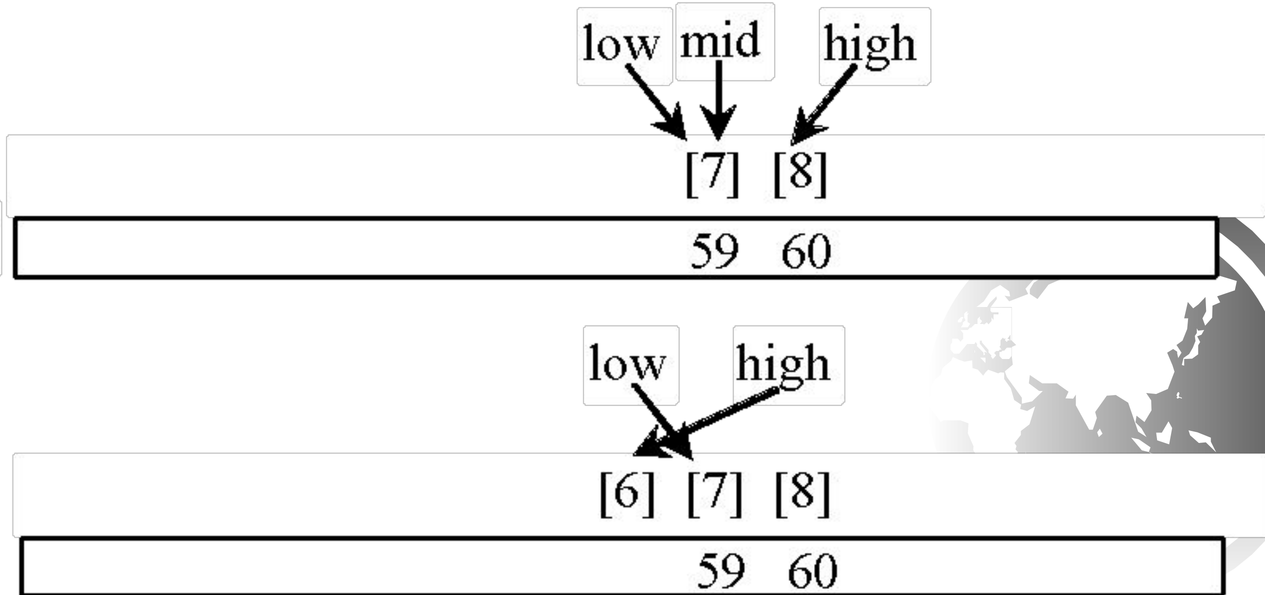
key < 66

list



key < 59

list



Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

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.

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

Sorting Arrays

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

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Why study sorting?

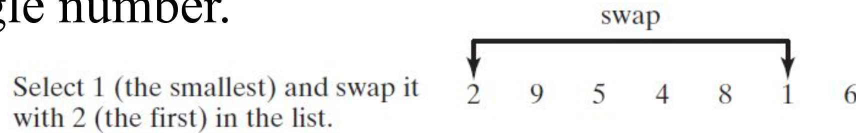
Sorting is a classic subject in computer science. There are three reasons for studying sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Selection Sort

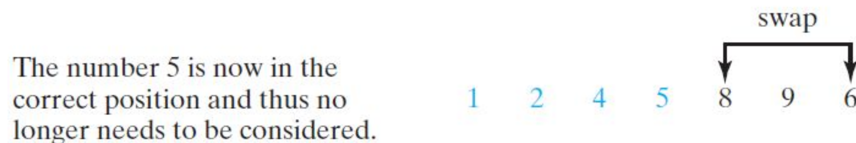
Selection sort finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it second, and so on until the list contains only a single number.



Select 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.



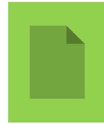
Select 6 (the smallest) and swap it with 8 (the first) in the remaining list.

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, the sort is completed.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Selection Sort Animation

http://www.cs.armstrong.edu/liang/animation/web/SelectionSort.html 

Exercise24_15: Selection Sort Animation

1 2 3 4 5 6 7 8 18 17 15 13 16 12 20 19 10 9 14 11

Step Reset

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+1..listSize-1]
}
```

```
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]
```

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

```
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+1; 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

```
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;
    }
  }
}
```

Invoke it

selectionSort(yourList)

Insertion Sort

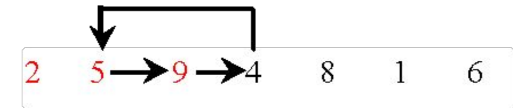
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted

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.

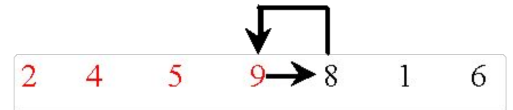
Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

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

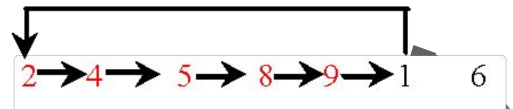
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.



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.



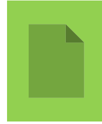
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

animation

Insertion Sort Animation

http://www.cs.armstrong.edu/liang/animation/web/InsertionSort.html 

InsertionSortAnimation

1 2 12 13 15 18 10 19 9 3 20 4 11 5 14 17 7 16 8 6

Step Reset

rights reserved.

animation

Insertion Sort

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

2	9	5	4	8	1	6

2	9	5	4	8	1	6

2	5	9	4	8	1	6

2	4	5	9	8	1	6

2	4	5	8	9	1	6

1	2	4	5	8	9	6

1	2	4	5	6	8	9

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.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
list	2	5	9	4				Step 1: Save 4 to a temporary variable currentElement
list	2	5		9				Step 2: Move list[2] to list[3]
list	2		5	9				Step 3: Move list[1] to list[2]
list	2	4	5	9				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
}
```

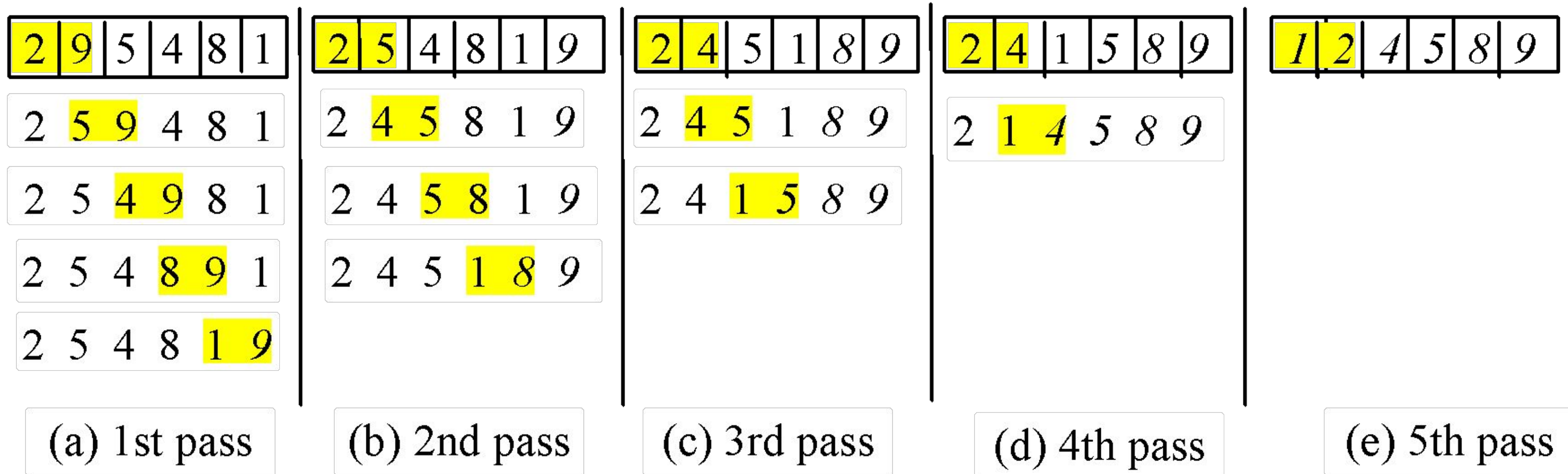
Expand

```
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;
```

InsertSort Run

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Bubble Sort



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

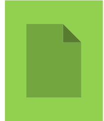
$$(n-1)+(n-2)+\ldots+2+1=\frac{n^2}{2}-\frac{n}{2}$$

BubbleSort

Run

Bubble Sort Animation

http://www.cs.armstrong.edu/liang/animation/web/BubbleSort.html



Bubble Sort Animation ×

www.cs.armstrong.edu/liang/animation/web/BubbleSort.html

Bubble Sort Animation by Y. Daniel Liang using JavaScript and Processing.js

Perform bubble sort for a list of 20 distinct integers from 1 to 20. Click the Step button to swap the first element in the remaining unsorted list with the smallest element in the remaining unsorted list. Click the Reset button to start over with a new random list.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	2	16	17	12	13	6	4	18	20	19	7	14	11	10	15	1	3	5	8	9

Step Reset

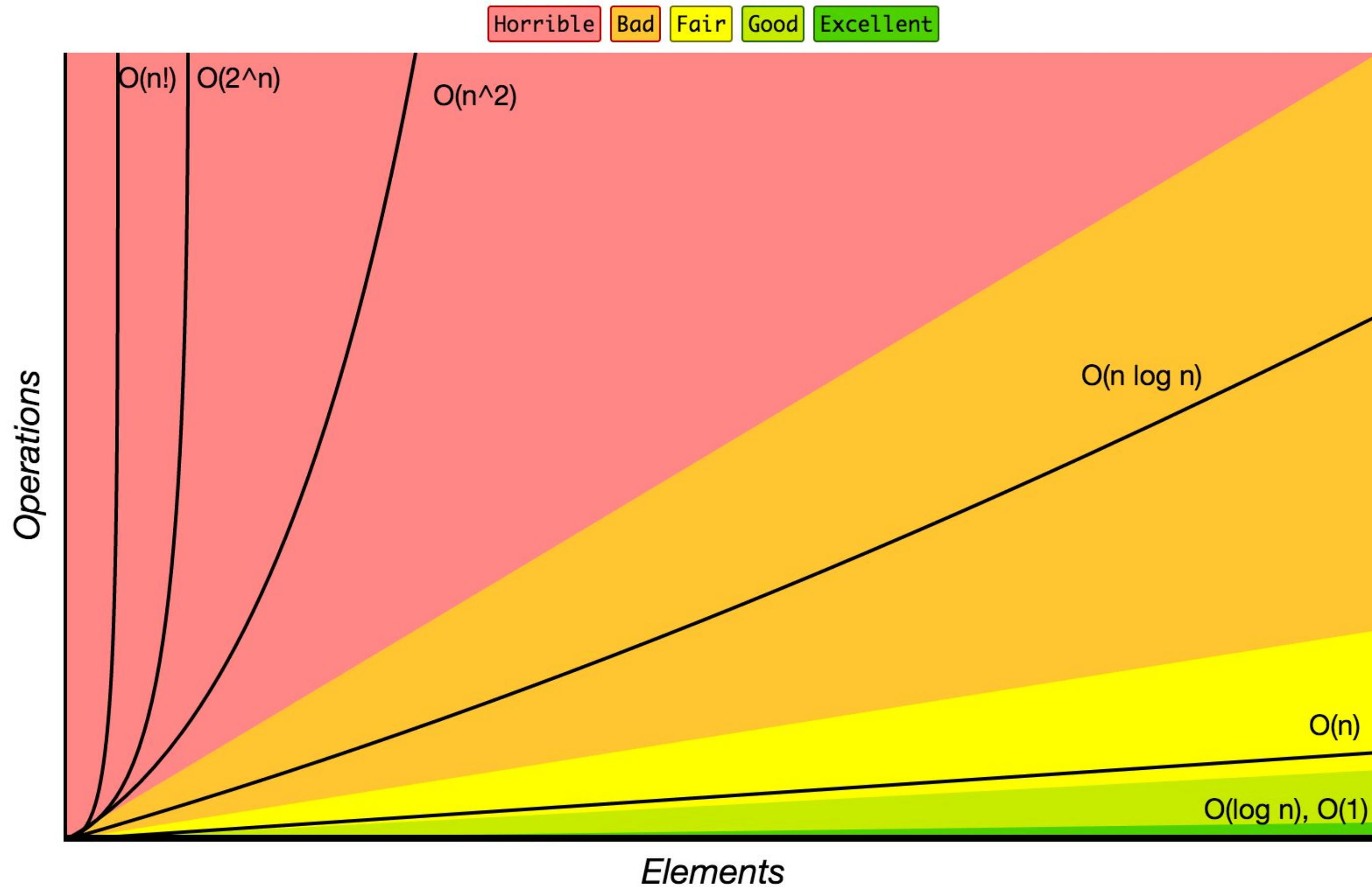
Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Computational Complexity (Big O)

- $T(n)=O(1)$ // constant time
- $T(n)=O(\log n)$ // logarithmic
- $T(n)=O(n)$ // linear
- $T(n)=O(n \log n)$ // linearithmic
- $T(n)=O(n^2)$ // quadratic
- $T(n)=O(n^3)$ // cubic

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Complexity Examples



http://bigocheatsheet.com/

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Complexity Examples

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

http://bigocheatsheet.com/

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Why does it matter?

Algorithm	10	20	50	100	1,000	10,000	100,000
O(1)	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s
O(log(n))	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s
O(n)	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s
O(n*log(n))	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s
$O(n^2)$	<1 s	<1 s	<1 s	<1 s	<1 s	2 s	3 m
$O(n^3)$	<1 s	<1 s	<1 s	<1 s	20 s	6 h	232 d
$O(2^n)$	<1 s	<1 s	260 d	∞	∞	∞	∞
O(n!)	<1 s	∞	∞	∞	∞	∞	∞
$O(n^n)$	3 m	∞	∞	∞	∞	∞	∞

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Misc Slides

Objectives

- To study and analyze time complexity of various sorting algorithms (§§23.2–23.7).
- To design, implement, and analyze insertion sort (§23.2).
- To design, implement, and analyze bubble sort (§23.3).
- To design, implement, and analyze merge sort (§23.4).

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

What data to sort?

The data to be sorted might be integers, doubles, characters, or objects. §7.8, "Sorting Arrays," presented selection sort and insertion sort for numeric values. The selection sort algorithm was extended to sort an array of objects in §11.5.7, "Example: Sorting an Array of Objects." The Java API contains several overloaded sort methods for sorting primitive type values and objects in the java.util.Arrays and java.util.Collections class. For simplicity, this section assumes:

- data to be sorted are integers,
- data are sorted in ascending order, and
- data are stored in an array. The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in an ArrayList or a LinkedList.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Don’t need to know Merge Sort for now

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

conquer

MergeSort

Run

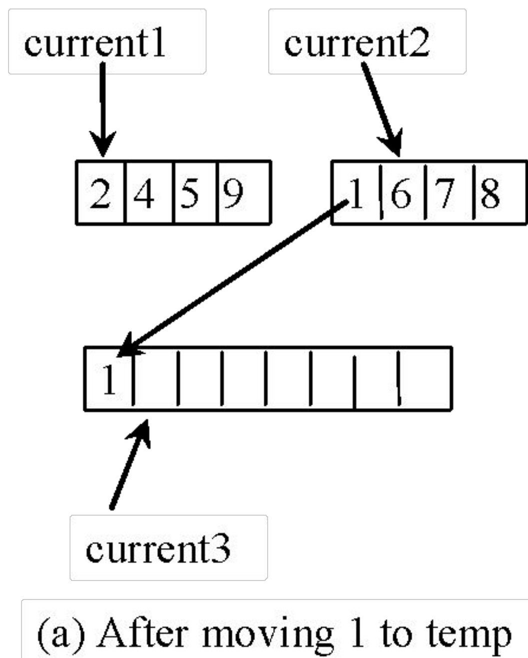
Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Merge Sort

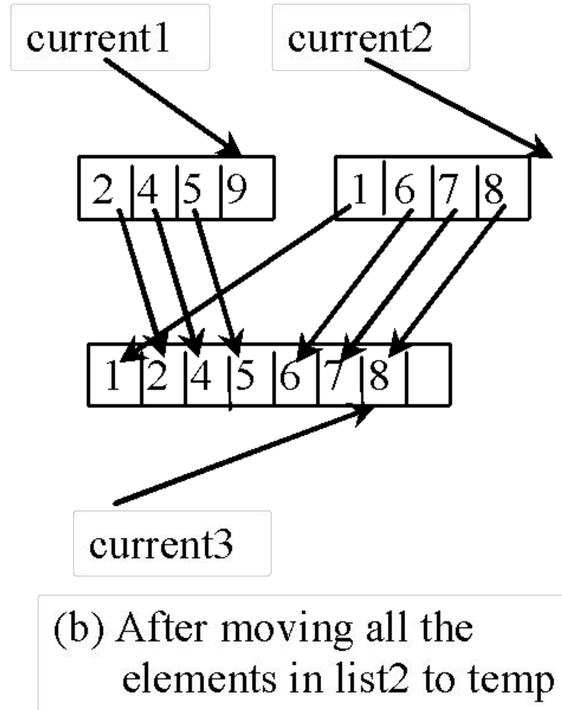
```
mergeSort(list):
    firstHalf = mergeSort(firstHalf);
    secondHalf = mergeSort(secondHalf);
    list = merge(firstHalf, secondHalf);
```

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

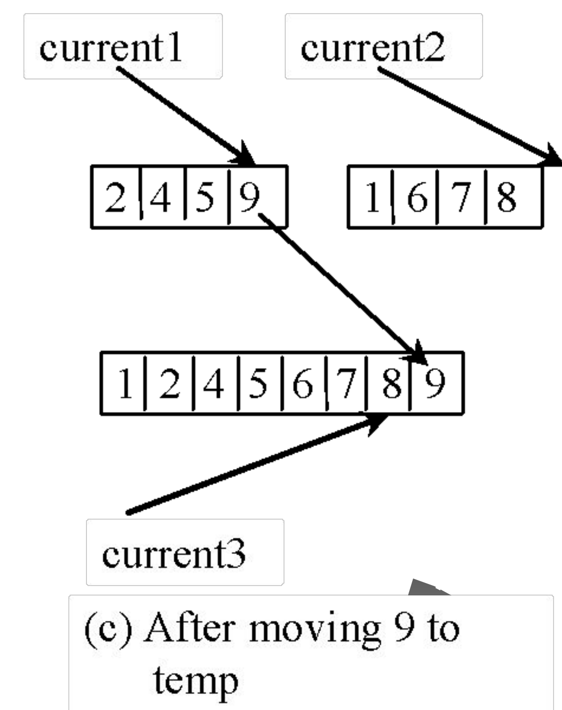
Merge Two Sorted Lists



(a) After moving 1 to temp



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



(c) After moving 9 to temp

Animation for Merging Two Sorted Lists

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

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,

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

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

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

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,

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

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

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

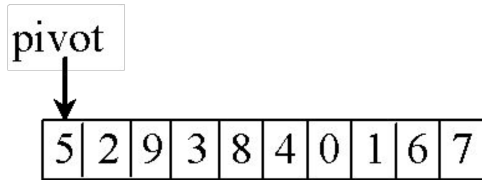
$$= n + 2n\log n - 2^{\log n} + 1 = 2n\log n + 1 = O(n\log n)$$

Quick Sort

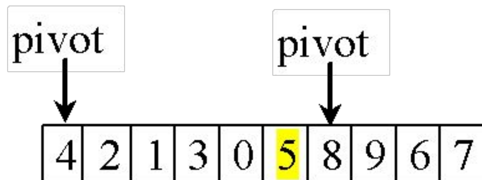
Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the *pivot*, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than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

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

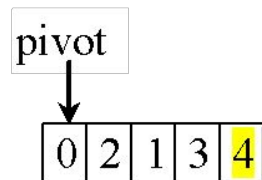
Quick Sort



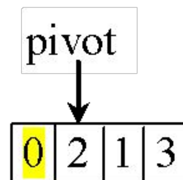
(a) The original array



(b)The original array is partitioned



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

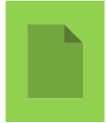


(d) The partial array (0 2 1 3) is partitioned

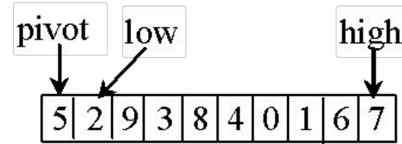
(e) The partial array (2 1 3) is partitioned

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

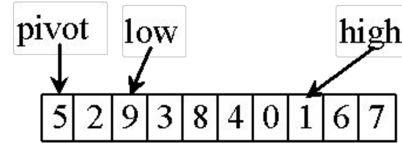
Partition



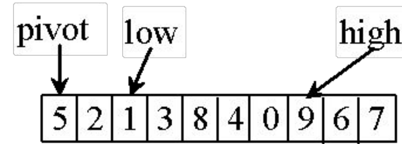
Animation for partition



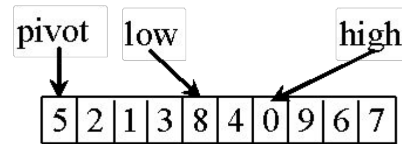
(a) Initialize pivot, low, and high



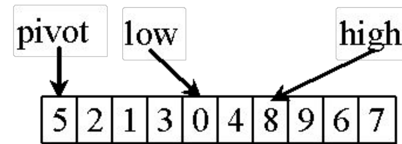
(b) Search forward and backward



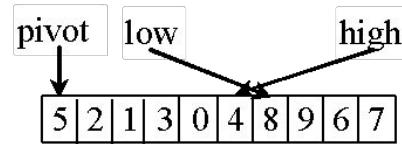
(c) 9 is swapped with 1



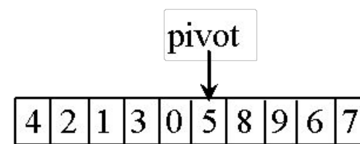
(d) Continue search



(e) 8 is swapped with 0



(f) when high < low, search is over



(g) pivot is in the right place

The index of the pivot is returned

QuickSort

Run

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Quick Sort Time

To partition an array of n elements, it takes $n-1$ comparisons and n moves in the worst case. So, the time required for partition is $O(n)$.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

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 $O(n^2)$ time:

$$(n-1)+(n-2)+...+2+1=O(n^2)$$

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,

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

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

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(nlogn)*. The exact average-case analysis is beyond the scope of this book.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.