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.

```java
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;
    }
}
```

<table>
<thead>
<tr>
<th>list</th>
<th>[0] [1] [2] ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>Compare key with list[i] for i = 0, 1, ...</td>
</tr>
</tbody>
</table>
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`. 
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
3   | 6 4 1 9 7 3 2 8
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;
}

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
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.
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

<table>
<thead>
<tr>
<th>Key</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1    2  3  4  6  7  8  9</td>
</tr>
<tr>
<td>8</td>
<td>1    2  3  4  6  7  8  9</td>
</tr>
<tr>
<td>8</td>
<td>1    2  3  4  6  7  8  9</td>
</tr>
<tr>
<td>8</td>
<td>1    2  3  4  6  7  8  9</td>
</tr>
</tbody>
</table>
Binary Search Animation

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

![Binary Search Animation](animation)
Binary Search, cont.

key is 11

key < 50

key > 7

key == 11
Binary Search, cont.

key is 54

key > 50

key < 66

key < 59
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.
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.
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.

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 4 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.

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.

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.
Selection Sort Animation

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

...
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]
}

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;
    }
}

Expand
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]
}

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

// Swap list[i] with list[currentMinIndex] if necessary;
Invoke it
selectionSort(yourList)
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.

Step 2: 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 Animation

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

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

<table>
<thead>
<tr>
<th>2</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

```java
1 2 4 5 8 9 6
```
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.

<table>
<thead>
<tr>
<th>list</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [1] [2] [3] [4] [5] [6]</td>
<td>2 5 9 4</td>
</tr>
</tbody>
</table>

Step 1: Save 4 to a temporary variable currentElement

<table>
<thead>
<tr>
<th>list</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [1] [2] [3] [4] [5] [6]</td>
<td>2 5 9</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>list</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [1] [2] [3] [4] [5] [6]</td>
<td>2 5 9</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>list</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [1] [2] [3] [4] [5] [6]</td>
<td>2 4 5 9</td>
</tr>
</tbody>
</table>

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]


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 
}

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

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

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

Big-O Complexity Chart

http://bigocheatsheet.com/

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

### Array Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Timsort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\Omega(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Tree Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n (\log(n))^2)$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$\Omega(nk)$</td>
<td>$\Theta(nk)$</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
</tr>
<tr>
<td>Cubesort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
</tbody>
</table>

http://bigocheatsheet.com/
Why does it matter?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
</tr>
<tr>
<td>O(log(n))</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
</tr>
<tr>
<td>O(n)</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
</tr>
<tr>
<td>O(n*log(n))</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
</tr>
<tr>
<td>O(n²)</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>2 s</td>
<td>3 m</td>
</tr>
<tr>
<td>O(n³)</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>20 s</td>
<td>6 h</td>
<td>232 d</td>
</tr>
<tr>
<td>O(2^n)</td>
<td>&lt;1 s</td>
<td>&lt;1 s</td>
<td>260 d</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>O(n!)</td>
<td>&lt;1 s</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>O(nⁿ)</td>
<td>3 m</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

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).
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.
Don’t need to know Merge Sort for now
Merge Sort

Diagram of Merge Sort algorithm with numbers and arrows indicating the divide and conquer process.
Merge Sort

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

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

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$
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,

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

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

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

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

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
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 \( O(n) \).
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:

$$(n - 1) + (n - 2) + \ldots + 2 + 1 = O(n^2)$$
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 $n$ elements using quick sort. So,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = O(n \log n)$$
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 $O(n \log n)$. The exact average-case analysis is beyond the scope of this book.