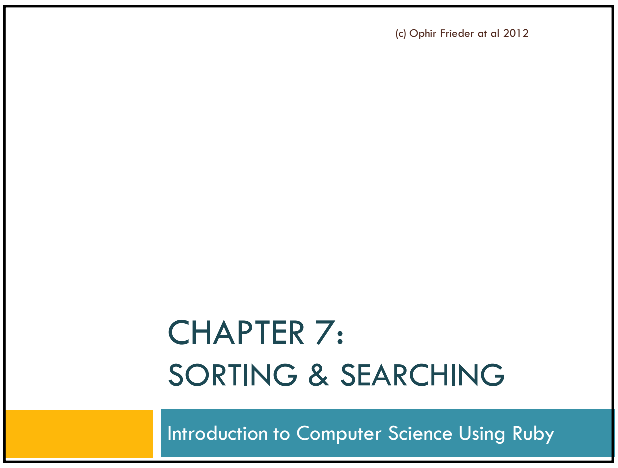
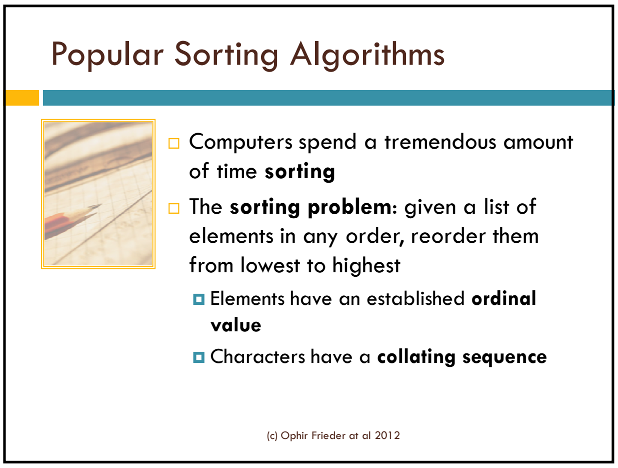


(c) Ophir Frieder at al 2012

CHAPTER 7:
SORTING & SEARCHING

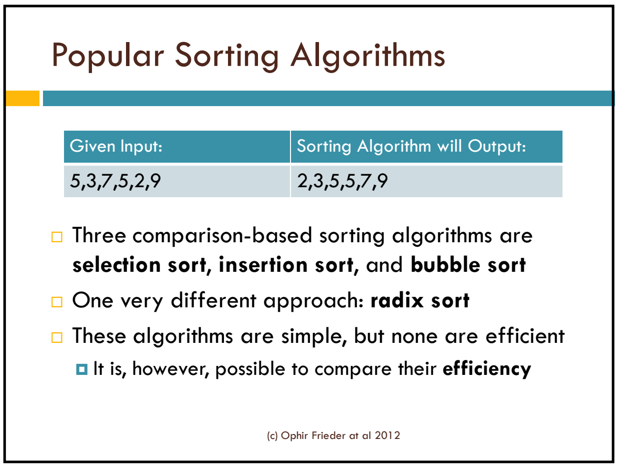
Introduction to Computer Science Using Ruby



Popular Sorting Algorithms

- Computers spend a tremendous amount of time **sorting**
- The **sorting problem:** given a list of elements in any order, reorder them from lowest to highest
  - Elements have an established **ordinal value**
  - Characters have a **collating sequence**

(c) Ophir Frieder at al 2012

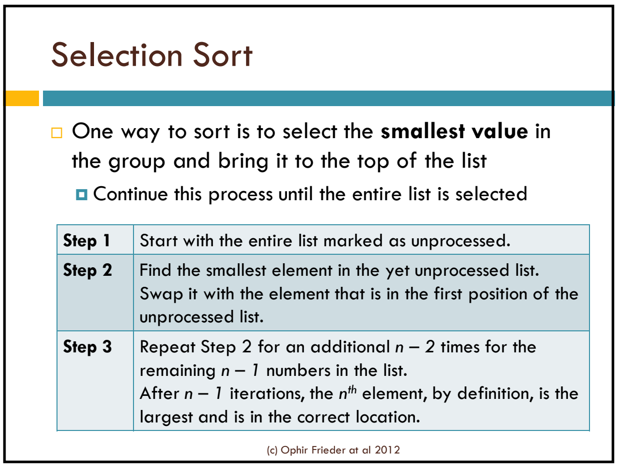


Popular Sorting Algorithms

Given Input:	Sorting Algorithm will Output:
5,3,7,5,2,9	2,3,5,5,7,9

- Three comparison-based sorting algorithms are **selection sort, insertion sort,** and **bubble sort**
- One very different approach: **radix sort**
- These algorithms are simple, but none are efficient
  - It is, however, possible to compare their **efficiency**

(c) Ophir Frieder at al 2012

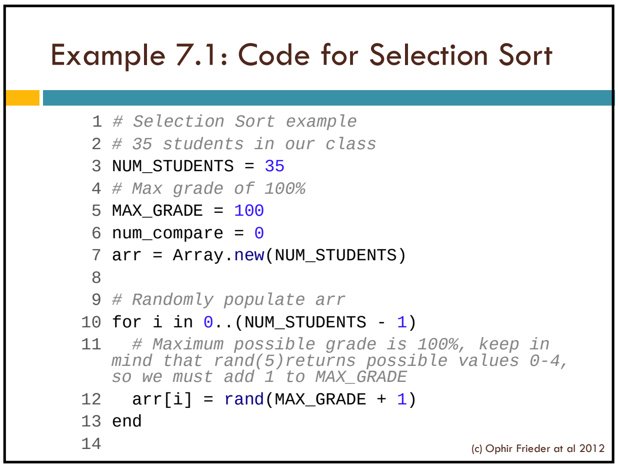


Selection Sort

- One way to sort is to select the **smallest value** in the group and bring it to the top of the list
  - Continue this process until the entire list is selected

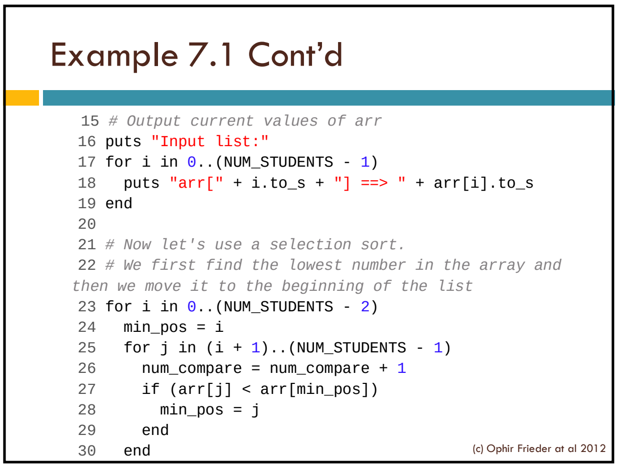
Step 1	Start with the entire list marked as unprocessed.
Step 2	Find the smallest element in the yet unprocessed list. Swap it with the element that is in the first position of the unprocessed list.
Step 3	Repeat Step 2 for an additional $n - 2$ times for the remaining $n - 1$ numbers in the list. After $n - 1$ iterations, the n^{th} element, by definition, is the largest and is in the correct location.

(c) Ophir Frieder at al 2012



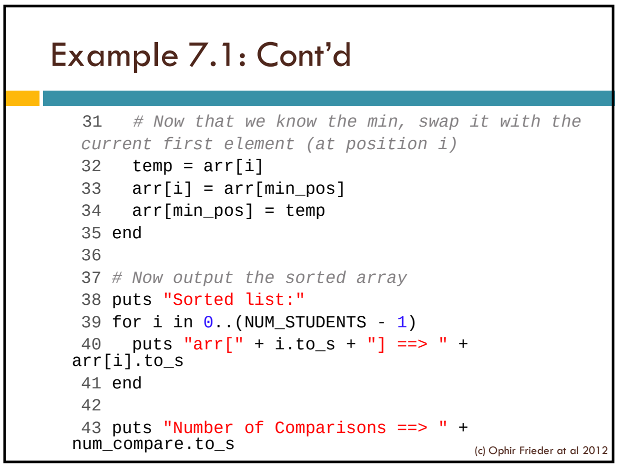
Example 7.1: Code for Selection Sort

```
# Selection Sort example
# 35 students in our class
NUM_STUDENTS = 35
# Max grade of 100%
MAX_GRADE = 100
num_compare = 0
arr = Array.new(NUM_STUDENTS)

# Randomly populate arr
for i in 0..(NUM_STUDENTS - 1)
  # Maximum possible grade is 100%, keep in mind that rand(5)returns possible values 0-4, so we must add 1 to MAX_GRADE
  arr[i] = rand(MAX_GRADE + 1)
end

```



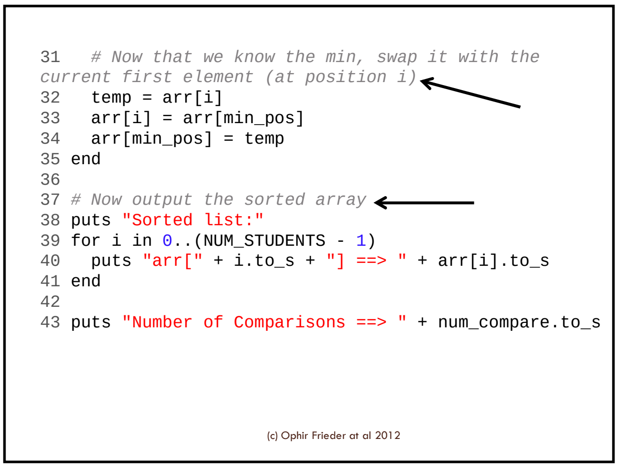
Example 7.1 Cont'd

```
# Output current values of arr
puts "Input list:"
for i in 0..(NUM_STUDENTS - 1)
  puts "arr[" + i.to_s + "] ==> " + arr[i].to_s
end

# Now let's use a selection sort.
# We first find the lowest number in the array and then we move it to the beginning of the list
for i in 0..(NUM_STUDENTS - 2)
  min_pos = i
  for j in (i + 1)..(NUM_STUDENTS - 1)
    num_compare = num_compare + 1
    if (arr[j] < arr[min_pos])
      min_pos = j
    end
  end
```



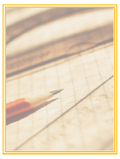
Example 7.1: Cont'd

```
  # Now that we know the min, swap it with the current first element (at position i)
  temp = arr[i]
  arr[i] = arr[min_pos]
  arr[min_pos] = temp
end

# Now output the sorted array
puts "Sorted list:"
for i in 0..(NUM_STUDENTS - 1)
  puts "arr[" + i.to_s + "] ==> " + arr[i].to_s
end

puts "Number of Comparisons ==> " + num_compare.to_s
```



```
  # Now that we know the min, swap it with the current first element (at position i)
  temp = arr[i]
  arr[i] = arr[min_pos]
  arr[min_pos] = temp
end

# Now output the sorted array
puts "Sorted list:"
for i in 0..(NUM_STUDENTS - 1)
  puts "arr[" + i.to_s + "] ==> " + arr[i].to_s
end

puts "Number of Comparisons ==> " + num_compare.to_s
```

Popular Sorting Algorithms: Insertion Sort



- Another way to sort is to start with a **new list**
  - Place each element into the list in order one at a time
  - The new list is always sorted

Popular Sorting Algorithms: Insertion Sort

Insertion sort algorithm:

- Step 1:
  - Consider only the **first element,** and thus, our list is sorted
- Step 2:
  - Insert the next element into the **proper position** in the already sorted list
- Step 3:
  - Repeat this process of adding **one new number for all *n* numbers**

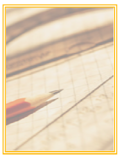
Example 7.2: Code for Insertion Sort

```
# Now let's use an insertion sort
# Insert lowest number in the array at the right place in the array
for i in 0..NUM_STUDENTS - 1
# Now start at current bottom and move toward arr[i]
  j = i
  done = false
  while ((j > 0) and (! done))
    num_compare = num_compare + 1
# If the bottom value is lower than values above it, swap it until it lands in a
 # place where it is not lower than the next item above it
    if (arr[j] < arr[j - 1])
      temp = arr[j - 1]
      arr[j - 1] = arr[j]
      arr[j] = temp
```

Example 7.2 Cont’d

```
    else
      done = true
    end
    j = j - 1
  end
end
```

Popular Sorting Algorithms: Bubble Sort



- Elements can be sorted based on **percolation**
  - Elements percolate to the right order by **swapping** neighboring elements
  - If the value is greater than the next, swap them
  - Small values “bubble” to the top of the list

Popular Sorting Algorithms: Bubble Sort

Bubble sort algorithm:

- Step 1:
  - Loop through all entries of the list
- Step 2:
  - For each entry, compare it to all **successive** entries
  - **Swap** if they are out of order

Example 7.3: Code for Bubble Sort

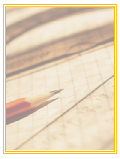
```
# Now let's use bubble sort. Swap pairs iteratively as we loop through the array
# From the beginning of the array to the second to last value
for i in 0..NUM_STUDENTS - 2
  # From arr[i + 1] to the end of the array
  for j in (i + 1)..NUM_STUDENTS - 1
    num_compare = num_compare + 1
    # If the first value is greater than the second value, swap them
    if (arr[i] > arr[j])
      temp = arr[j]
      arr[j] = arr[i]
      arr[i] = temp
    end
  end
end
```

```
# Now let's use bubble sort. Swap pairs iteratively as we loop through the array
# From the beginning of the array to the second to last value
for i in 0..NUM_STUDENTS - 2
  # From arr[i + 1] to the end of the array
  for j in (i + 1)..NUM_STUDENTS - 1
    num_compare = num_compare + 1
    # If the first value is greater than the second value, swap them
    if (arr[i] > arr[j])
      temp = arr[j]
      arr[j] = arr[i]
      arr[i] = temp
    end
  end
end
```

Complexity Analysis

- To evaluate an algorithm, analyze its **complexity**
  - Count the number of steps involved in executing the algorithm
- How many units of time are involved in processing ***n*** elements of input?
  - Need to determine the **number of logical steps** in a given algorithm

Complexity Analysis: Family of Steps

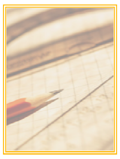


- Addition and subtraction
- Multiplication and division
- Nature and number of loops controls

Complexity Analysis: Family of Steps

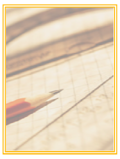
- Count how many steps of each family are required for *n* operations like $a^2 + ab + b^2$
  - This statement has **3*n* multiplications** and **2*n* additions**
- Can compute the same expression using $(a + b)^2 – ab$
  - This has ***2n* multiplications** and ***2n* additions**
  - This expression is better than the original
  - For very large values of *n*, this may make a significant difference in computation

Complexity Analysis



- For complexity analysis, **forgo constants**
  - $(n – 1)$ and n have no difference in terms of complexity
- Assume that all computations are of the **same family of operations**

Complexity Analysis



- Consider the **three comparison-based** sorting algorithms
- For all, the outer loop has ***n*** **steps**
- For the inner loop, the size of the list **shrinks** or **increases,** by one with each pass.

Complexity Analysis

- The first step is n, the next $n - 1$, and so forth
- Add 1 to the sum, and it becomes an **arithmetic series:** $\frac{n(n + 1)}{2}$

Complexity Analysis

- The total number of steps for these algorithms are:

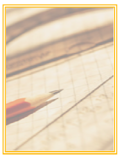
  $$\frac{n(n + 1)}{2} - 1$$

- Complexity is considered $O(n^2)$
  - It is not exact, but simply an **approximation**
  - The dominant portion of this sum is n^2

Complexity Analysis

- There is a **best, average,** and **worst** case analysis for computations
- For Selection and Bubble Sort algorithms, all cases are the same; the processing is **identical**
- For Insertion Sort, processing an already sorted list will be $O(n)$ → best case scenario
- A list needing to be completely reversed will require $O(n^2)$ steps → worst case scenario
  - Average case is the same

Complexity Analysis

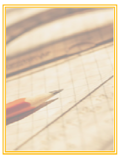


- Radix Sort works in **O(*dn*)**
  - *d* is the number of digits that need **processing**
  - *n* is the number of entries that need **sorting**
- Radix Sort works faster than the other examples
- Other algorithms that run in **O(*n*log(*n*))**:
  - quicksort
  - *mergesort*
  - *heapsort*

Searching

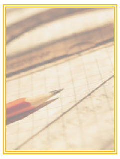
- **Searching** is common task computers perform
- Two parameters that affect search algorithm selection:
  1. Whether the list is **sorted**
  2. Whether all the elements in the list are **unique** or have **duplicate** values
- For now, our implementations will assume there are **no duplicates** in the list
- We will use two types of searches:
  - **Linear search** for unsorted lists
  - **Binary search** for sorted lists

Searching: Linear Search



- The simplest way to find an element in a list is to check if it **matches** the sought after value
  - Worst case: the entire list must be linearly searched
  - This occurs when the value is in the last position or not found

Searching: Linear Search



- The average case requires searching half of the list
- The best case occurs when the value is in the first element in the list

Searching: Linear Search

Linear Search Algorithm:

```
for all elements in the list do
    if element == value_to_find then return position_of (element)
    end # if
end    # for
```

Consider using this search on a list that has **duplicate elements**

- You cannot assume that once one element is found, the search is done
- Thus, you need to continue searching through the **entire list**

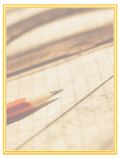
Example 7.5: Code for Linear Search

```ruby
# Example Linear Search
NUM_STUDENTS = 35
MAX_GRADE = 100
arr = Array.new(NUM_STUDENTS)
value_to_find = 8
i = 1
found = false

# Randomly put some student grades into arr
for i in 0..NUM_STUDENTS - 1
  arr[i] = rand(MAX_GRADE + 1)
end

puts "Input List:"
for i in 0..NUM_STUDENTS - 1
  puts "arr[" + i.to_s + "] ==> " + arr[i].to_s
end

```

Example 7.5 Cont'd

```ruby
# Loop over the list until it ends, or we have found
our value
while ((i < NUM_STUDENTS) && (not found))
  # We found it :)
  if (arr[i] == value_to_find)
    puts "Found " + value_to_find.to_s + " at position " +
i.to_s + " of the list."
    found = true
  end
  i = i + 1
end

# If we haven't found the value at this point, it doesn't
exist in our list
if (not found)
  puts "There is no " + value_to_find.to_s + " in the
list."
end
```

Searching: Binary Search



- For binary search, begin searching at the **middle** of the list
  - If the item is less than the middle, check the middle item between the first item and the middle
  - If it is more than the middle item, check the middle item of the section between the middle and the last section
- The process stops when the value is found or when the remaining list of elements to search consists of one value

Searching: Binary Search

- Following this process reduces half the search space
- The algorithm is an $O(\log_2(n))$
  - Equivalent to $O(\log(n))$
  - This is the same for the average and worst cases

Searching: Binary Search

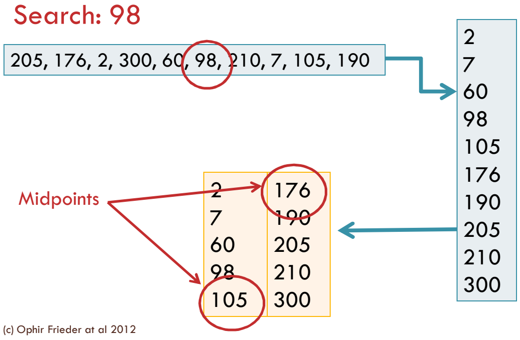
- Keep in mind that a binary search requires an **ordered list**
  - An unsorted list needs to be sorted before the search
    - If the search occurs rarely, you should not sort the list
    - If the list is updated infrequently, sort and then search the list
- Check values immediately preceding and following the current position to modify the search to work with **duplicates**

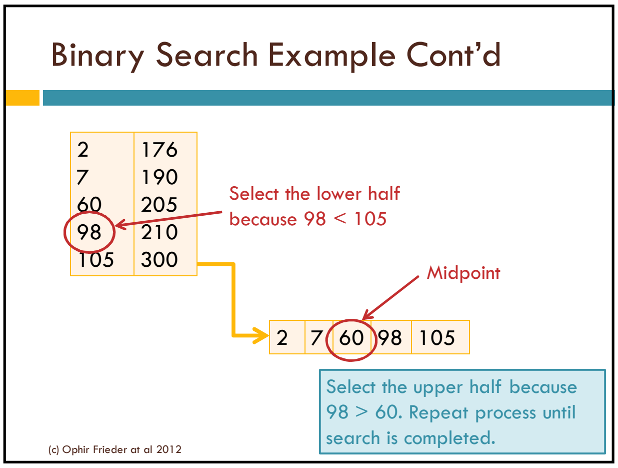
Binary Search Example

1. Create an ordered list
2. Divide entries into 2 halves
3. Locate midpoint(s) and determine if number is below or above midpoint(s)
4. Repeat steps 2 and 3 until search is completed

Binary Search Example

Search: 98



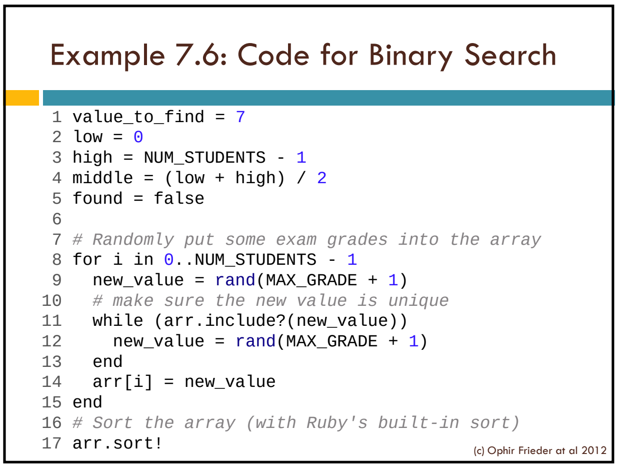


Binary Search Example Cont'd

Select the lower half because 98 < 105

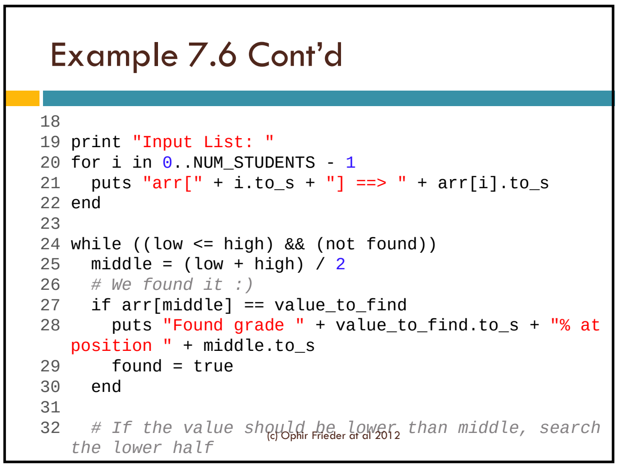
Midpoint

Select the upper half because 98 > 60. Repeat process until search is completed.



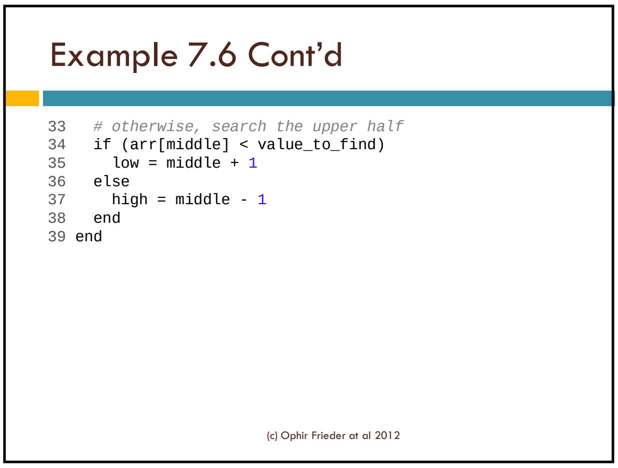
Example 7.6: Code for Binary Search

```ruby
value_to_find = 7
low = 0
high = NUM_STUDENTS - 1
middle = (low + high) / 2
found = false

# Randomly put some exam grades into the array
for i in 0..NUM_STUDENTS - 1
  new_value = rand(MAX_GRADE + 1)
  # make sure the new value is unique
  while (arr.include?(new_value))
    new_value = rand(MAX_GRADE + 1)
  end
  arr[i] = new_value
end
# Sort the array (with Ruby's built-in sort)
arr.sort!
```



Example 7.6 Cont'd

```ruby

print "Input List: "
for i in 0..NUM_STUDENTS - 1
  puts "arr[" + i.to_s + "] ==> " + arr[i].to_s
end

while ((low <= high) && (not found))
  middle = (low + high) / 2
  # We found it :)
  if arr[middle] == value_to_find
    puts "Found grade " + value_to_find.to_s + "% at position " + middle.to_s
    found = true
  end

  # If the value should be lower than middle, search the lower half
```



Example 7.6 Cont'd

```ruby
  # otherwise, search the upper half
  if (arr[middle] < value_to_find)
    low = middle + 1
  else
    high = middle - 1
  end
end
```

```
  # otherwise, search the upper half
  if (arr[middle] < value_to_find)
    low = middle + 1
  else
    high = middle - 1
  end
end
```

Summary

- Sorting is a problem that occurs in many applications in computer science
- **Comparison-based sorting** simply compares the items to determine the order
- **Radix Sort** sorts without directly comparing

Summary

- Computer scientists use **complexity analysis** to discuss algorithm performance
- Searching can be done by **linear search**
- Binary search can be used if the list is **sorted**
- Know the difference in complexity between linear and binary searches