Lecture 13

Sorting III

- Lab 4 — Preview
- Generalized Sorting
  - Generics
  - Comparators
- Midterm Review
  - Problem 3
  - Problem 4
Lab 4 — Preview
Seattle Pet Licenses
Pet licenses issued by the Seattle Animal Shelter between 2005 and early 2017

License
Database: Open Database, Contents: © Original Authors

Tags
social issues and advocacy, animals

Description

Context
The city of Seattle makes available its database of pet licenses issued from 2005 to the beginning of 2017 as part of the city's ongoing Open Data Initiative. The data is also obtainable from the Socrata Open Data Access (SODA) portal in either CSV or JSON formats. It is also made available here (unofficially, I have no official affiliation with the city of Seattle or the Seattle Animal Shelter) to help spread awareness of the dataset and Seattle's Pet Licensing initiative.

Content
Seattle Pet Licenses Dataset

Data Explorer
4.76 MB

Seattle Pet Licenses in .csv (comma separated values) format.
<table>
<thead>
<tr>
<th>Date</th>
<th>SeattlePets.csv</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 14, 2016</td>
<td>S117962, Misty</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>June 11, 2016</td>
<td>S117962, Frankie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>June 8, 2016</td>
<td>S117962, Jones</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>August 8, 2016</td>
<td>S117962, Minnie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>August 11, 2016</td>
<td>S117962, Charlie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>August 16, 2016</td>
<td>S117962, Lucy</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 12, 2016</td>
<td>S117962, Lydia</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 15, 2016</td>
<td>S117962, Max</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 18, 2016</td>
<td>S117962, Charlie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 21, 2016</td>
<td>S117962, Lucy</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 14, 2016</td>
<td>S117962, Minnie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 17, 2016</td>
<td>S117962, Dave</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>November 20, 2016</td>
<td>S117962, Max</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 1, 2016</td>
<td>S117962, Charlie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 4, 2016</td>
<td>S117962, Lucy</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 7, 2016</td>
<td>S117962, Max</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 10, 2016</td>
<td>S117962, Minnie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 13, 2016</td>
<td>S117962, Dave</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 16, 2016</td>
<td>S117962, Max</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 19, 2016</td>
<td>S117962, Charlie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 22, 2016</td>
<td>S117962, Lucy</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 25, 2016</td>
<td>S117962, Minnie</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 28, 2016</td>
<td>S117962, Dave</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
<tr>
<td>December 31, 2016</td>
<td>S117962, Max</td>
<td>Cat, Domestic Shorthair, 91217</td>
</tr>
</tbody>
</table>

SeattlePets.csv is part of Lab 4.
// A simple class holding information about a pet.

import structure5.*;
import java.util.Scanner;
import java.io.*;

/**
 * A class for keeping track of information about a pet:
 * name, species, breed, license, license date, and zipcode.
 */

public class Pet
{
    /**
     * Construct a pet from a Vector generated from the CSV reader on the
     * Seattle pet registration database.
     * Field  Meaning
     * 0  Licensing date
     * 1  License ID (alphanumeric)
     * 2  Name
     * 3  Species (e.g. "Cat")
     * 4  Breed (primary; e.g. "retriever")
     * 5  Breed (secondary; ignored; e.g. "mix")
     * 6  Zipcode (a possibly empty integer string representation)
     * Secondary breed information is ignored, and integer zipcode defaults to 0.
     * @param description A Vector of strings in Seattle Pet format
     */
    public Pet(Vector<String> description) {
    }

    /**
     * Returns the license date associated with the pet. E.g. "June 08 2018"
     * @return A string representing the licensing date.
     */
    public String licenseDate()
    {
        return "";
    }
}
/ A stable sort - insertion sort - from the Java Structures text.

public static void insertionSort(int data[], int n)
// pre: 0 <= n <= data.length
// post: values in data[0..n-1] are in ascending order
{
    int numSorted = 1; // number of values in place
    int index; // general index
    while (numSorted < n) {
        // take the first unsorted value
        int temp = data[numSorted];
        // ...and insert it among the sorted:
        for (index = numSorted; index > 0; index--) {
            if (temp < data[index-1]) {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        // reinsert value
        data[index] = temp;
        numSorted++;
    }
}
Generalized Sorting
Discussion: Generalized Sorting

When discussing sorting algorithms we have been focused on int arrays.

- How would we sort other common objects like String?
- How would we sort new objects like Pet?

How is this handled in Java? (Or how should it be handled in Java?)

Think about this for 2 minutes.
Then discuss it with your neighbor for 3 minutes.

Is there only one way of sorting a given type of object?

- Provide five different ways of sorting String objects. Be creative!
- Provide five different ways of sorting Pet objects.

How is this handled in Java? (Or how should it be handled in Java?)
Generics, Comparables, Comparators, Lambdas
Activity: Generalized Sorting

Determine answers to the following questions to the best of your abilities:

- What is a **generic**?
- What is a **comparable**? What is a **comparator**?
- What is a **lambda expression**? (Also know as a **lambda function**.)

Work with your neighbor for 5 minutes.

Relate what you have found to the previously discussed problems.

- How could you sort **String** objects in five different ways?
- How could you sort **Pet** objects in five different ways?

You will learn more about the precise mechanisms for doing these things in Java in Lab 4.
Midterm Review
Problem 3 [15 points]
Big-O and Counting.

a. A subsequence of characters found in a string, str, is any string that can be constructed by possibly deleting characters from str. Two subsequences are distinct if they are not equal. For example, there are 7 distinct subsequences of "eve":

"", "e", "v", "ev", "ee", "ve", "eve"

If str has length n, then what is the maximum number of distinct subsequences that it can contain? What is the minimum number? Provide exact answers to these two questions (e.g., the maximum is $n^2 + 1$), as opposed to a big-O.

b. Our formal definition of big-O is below. Fill in the blanks. Also provide an intuitive definition of big-O in your own words, making sure to address the role of $c$ and $n_0$.

Formal definition:

**Definition 5.1** A function $f(n)$ is $O(g(n))$ (read “order $g$” or “big-O of $g$”), if and only if there exist two positive constants, $c$ and $n_0$, such that

\[
\text{for all } n \geq n_0.
\]

Intuitive definition:

c. Rank the following from smallest to largest. Provide your answer in the table below. For example, $O(1)$ is the smallest, so it is given rank 1, as indicated in the table. You do not need to justify your answer to this question.

<table>
<thead>
<tr>
<th>$O(n)$</th>
<th>$O(n^2)$</th>
<th>$O(2^n)$</th>
<th>$O(n^3)$</th>
<th>$O(1)$</th>
<th>$O(n!)$</th>
<th>$O(log(n))$</th>
<th>$O(n^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d. Suppose that A is an int array of length n that is sorted from smallest to largest. How quickly could you determine the number of distinct values in the array? For example, if A is $\{1, 1, 3, 4, 4\}$ then it has three distinct values. Provide your answer in big-O along with a brief justification.

e. The following function binary(n) takes a non-negative integer, n, and returns a string that gives the base-2 (binary) representation of n. What is its run-time in big-O?

```java
/**
 * Build a String with the binary representation of a number n.
 * @param number the number to represent in binary
 * @result a String whose characters are the binary digits of n
 * @pre
 */
public static String binary(int n) {
    if (n < 2) return ""+n;
    else return binary(n/2)+n%2);
}
```
Problem 4  [20 points]
Debugging. A ternary search is similar to a binary search, except that the search space is repeatedly divided into thirds instead of halves. You are trying to implement ternary search in a function \texttt{find} which calls a recursive function \texttt{findr} (see next page). The input to \texttt{find} is an int array \texttt{A} that is sorted from smallest to largest, and a target value \texttt{t}. If the target is in \texttt{A}, then \texttt{find} should return an index with \texttt{A[index]} = \texttt{t}; otherwise, it should return null.

Good news: \texttt{find} is partially working! Bad news: It has some bugs. To debug your code, you created an array \texttt{A} = \{1,1,3,3,5,7,9,11,11\} and tested \texttt{find} with all targets \texttt{t} between 0 and 12, with results given in the table below, where \texttt{x} denotes infinite recursion.

<table>
<thead>
<tr>
<th>target \texttt{t}</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>expected</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>received</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Running \texttt{find(t, A)} on array \texttt{A} = \{1,1,3,3,5,7,9,11,11\} with targets \texttt{t} = 0, 1,\ldots, 12.
For example, when the target \texttt{t} = 5, the \texttt{find} function is correct in returning \texttt{4} since \texttt{A[4]} = \texttt{5}.
Similarly, when the target is \texttt{t} = 8, the \texttt{find} function is correct in returning \texttt{null} since \texttt{A} has no \texttt{8}.

a. Your tests have identified some false negatives, including \texttt{find}(7, A) returning \texttt{null}.
You believe that these errors are being caused by the recursive calls in lines 34–46.
To test this hypothesis, consider the first level of recursion resulting from \texttt{find(t, A)}, where \texttt{A} is the test array given above. Depending on the target \texttt{t}, a recursive call will be made on line 36, 40, or 44. Provide the missing arguments for these three calls in the figure below. For example, line 36 calls \texttt{findr(t, A, left, middle1 - 1)}, so you should provide the values of \texttt{left} and \texttt{middle1} - 1 in the space provided.
You may find it helpful to record intermediate values directly in the code. For example, you may wish to write \texttt{size = 9} on line 17, since \texttt{right - left + 1 = 8 - 0 + 1 = 9}.

\begin{verbatim}
find(t, A)
findr(t, A, 0, 8)
findr(t, A, , )
findr(t, A, , )

line 36 line 40 line 44
\end{verbatim}

Function calls resulting from testing \texttt{find} on array \texttt{A} = \{1,1,3,3,5,7,9,11,11\}.
Depending on the target \texttt{t}, one of the three recursive calls on the bottom row will be called.
Fill in the missing \texttt{left} and \texttt{right} arguments for each of these calls with specific values.

b. Explain how the three recursive calls from part a. lead to false negatives in your tests. Are there any other issues in lines 34–46 that would lead to false negatives?

c. Your tests have also identified infinite recursion, including \texttt{findr}(9, A) running forever. In general, what are some potential causes of infinite recursion in implementations of binary search and ternary search?

d. You believe that your infinite recursion is caused by the base cases in lines 7–13 and/or the calculations in lines 23–25. Identify the issue. Note that your assertion on line 26 is never triggered. Hint: Think about \texttt{size1} and the recursive call in line 36.
public static Integer findr(int t, int A[]) {
    return findr(t, A, 0, A.length - 1);
}

protected static Integer findr(int t, int A[], int left, int right) {
// Base Case: There are no elements to search.
    if (left > right) return null;

// Base Case: There is one element to search.
    if (left == right) {
        if (t == A[left]) return left;
        else return null;
    }

// Compute the size of the range A[left], ..., A[right], and
// divide it into thirds, and account for any extra remaining.
    int size = right - left + 1;
    int third = size / 3; // Java rounds towards zero.
    int extra = size % 3; // The remainder modulo 3.

// Split size into three parts, with the extra in the first part.
// Assert that the three smaller sizes sum to the overall size.
    int size1 = third + extra;
    int size2 = third;
    int size3 = third;
    Assert.condition(size1 + size2 + size3 == size, "Wrong size sum");

// We will split A[left], ..., A[right] into three sub-ranges.
// Compute the two middle points.
    int middle1 = left + size1;
    int middle2 = left + size1 + size2;

// Search for the target value in one of the three sub-ranges.
    if (t < A[middle1]) {
        return findr(t, A, left, middle1 - 1);
    } else if (t < A[middle2]) {
        return findr(t, A, middle1, middle2 - 1);
    } else {
        return findr(t, A, middle2, right - 1);
    }
}