Search Algorithms & Recursion

COMP 110
Summer II 2012

Peter Lincoln
7/18/2012
Questions?

- Yesterday:
  - Inheritance: Abstract Classes & Interfaces
  - Inheritance Exercise
Today in COMP 110

- Lecture:
  - Search Algorithms & Recursion

- Lab 8
Arrays Review

- Recall that arrays contain a list of elements, all of the same type, e.g.:
  - `int[] scores = {7, 5, 2, 9};`
  - `double[] temperatures = new double[20];`
  - `String[] names = new String[9];`
  - `Student[] students = new Student[33];`

- Arrays created with the `new` operator
  - Primitive arrays are filled with zeros, '\0', or `false` (depending on type)
  - Object arrays are filled with `null`
Arrays Review

- We can index the arrays using the [] notation:
  - `int someScore = scores[5];`
  - `names[7] = "Fred Jones";`

- Processing arrays usually involves loops
  - Typically for loops for read/write operations
  - Can use for each loops for reading when the index does not matter
Arrays Review

Scanner keyboard = new Scanner(System.in);

System.out.println("Enter 5 basketball scores:");
int[] scores = new int[5];
int scoreSum = 0;
for (int i = 0; i < scores.length; i++)
{
    scores[i] = keyboard.nextInt();
    scoreSum += scores[i];
}

double average = (double) scoreSum / scores.length;
System.out.println("Average score: "+ average);

for (int i = 0; i < scores.length; i++)
{
    if (scores[i] > average)
        System.out.println(scores[i] +": above average");
    else if (scores[i] < average)
        System.out.println(scores[i] +": below average");
    else
        System.out.println(scores[i] +": equal to the average");
}
Arrays Review

- Printing 1-D Arrays
  - public static void printArrayLn(int[] array)
    {
      for (int i = 0; i < array.length; i++)
      {
        if (i > 0)
          System.out.print(" ");

        System.out.print(array[i]);
      }

    System.out.println();
    }

Searching for an Element

- Suppose we have an array
  - How can we find the index of a particular value?
/**
 * Searches the specified array for the specified value.
 * @param array an array to search
 * @param target the value to find
 * @return the first index of the value or -1 if the value was not found
 */

public static int findIndex(int[] array, int target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return i;
        }
    }
    return -1;
}
What if the array is sorted?

- Do we need to start at the beginning and look all the way through?

- What if we start in the middle of the array
  - `int mid = (first + last) / 2;`

- Since the array is sorted
  - If `array[mid] > target`, we know that `target` must be in the first half of the array
  - Else if `array[mid] < target`, then `target` must be in the second half of array
  - Else (`array[mid] == target`), return `mid` as the index of `target`

- This is called *binary search*
public static int binarySearch(int[] sortedArray, int target) {
    int first = 0;
    int last = sortedArray.length - 1;
    while (first <= last) {
        int mid = (first + last) / 2;
        if (target == sortedArray[mid]) {
            // the target is at mid, so return mid
            return mid;
        } else if (target < sortedArray[mid]) {
            // the target is before mid, so search again in the subrange before mid
            last = mid - 1;
        } else { // (target > sortedArray[mid])
            // the target is after mid, so search again in the subrange after mid
            first = mid + 1;
        }
    }
    // no more indices left to check, so report failure (-1)
    return -1;
}
Executing Binary Search

<table>
<thead>
<tr>
<th>first</th>
<th>mid</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>sortedArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

target = 5

target < sortedArray[mid]
Executing Binary Search

target = 5

first    mid    last

<table>
<thead>
<tr>
<th>sortedArray</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>21</td>
</tr>
</tbody>
</table>

target > sortedArray[mid]
Executing Binary Search

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>sortedArray</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

```target = 5

mid

first

last

```

target == sortedArray[mid]

Success! Return 2
Executing Binary Search

target = 13

```
<table>
<thead>
<tr>
<th>first</th>
<th>mid</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
target > sortedArray[mid]
```
Executin Binary Search

target = 13

<table>
<thead>
<tr>
<th>first</th>
<th>mid</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>sortedArray</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

target < sortedArray[mid]
Executing Binary Search

target = 13

<table>
<thead>
<tr>
<th>sortedArray</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>21</td>
</tr>
</tbody>
</table>

target > sortedArray[mid]
Executing Binary Search

target = 13

<table>
<thead>
<tr>
<th>sortedArray</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>21</td>
</tr>
</tbody>
</table>

first > last

Failure! Return -1
Divide & Conquer

- We saw this when designing methods (Top-Down Design)
  - Start with a big problem
  - Decompose problem into smaller subtasks
  - Solve subtasks to solve big problem
Divide & Conquer

- Binary search is an example of a divide and conquer algorithm
  - Start with a big array
  - Check if the center element is what we want
  - If not, then divide the array in half, and process the half that can contain the answer
    - Start with the half array
    - Check if its center is what we want
    - If not, then divide the array in half, ...

- Note that the subtask is almost the same as the original task
Recursive Algorithm

- A *recursive algorithm* is an algorithm that divides up the problem into smaller versions of itself.
  - The algorithm makes recursive calls upon itself using smaller inputs.

- Binary search can be implemented as a recursive algorithm.
Recursive Binary Search

```java
public static int recursiveBinarySearch(int[] sortedArray, int target)
{
    return recursiveBinarySearch(sortedArray, target, 0, sortedArray.length - 1);
}

private static int recursiveBinarySearch(int[] sortedArray, int target, int first, int last)
{
    if (first > last)
    {
        // no more indices left to check, so report failure (-1)
        return -1;
    }
    else
    {
        int mid = (first + last) / 2;
        if (target == sortedArray[mid])
        {
            // the target is at mid, so return mid
            return mid;
        }
        else if (target < sortedArray[mid])
        {
            // the target is before mid, so search again in the subrange before mid
            return recursiveBinarySearch(sortedArray, target, first, mid - 1);
        }
        else  // (target > sortedArray[mid])
        {
            // the target is after mid, so search again in the subrange after mid
            return recursiveBinarySearch(sortedArray, target, mid + 1, last);
        }
    }
}
```
Recursive Binary Search Demo

- In Eclipse

- Source code is available on the course website
Components of a Recursive Method

- There is a branching statement that leads to different cases
  - At least one branch includes a recursive call to the method
    - Recursive cases
  - At least one branch includes no recursive call and returns (a value)
    - Base cases or stopping cases
A Simpler Example: Factorial

- factorial(n) is the product of the first n numbers
  - factorial(0) is 1
  - factorial(1) is 1
  - factorial(2) is 2 (2 * 1)
  - factorial(3) is 6 (3 * 2 * 1)
  - factorial(4) is 24 (4 * 3 * 2 * 1)
A Simpler Example: Factorial

- factorial(n) is the product of the first n numbers
  - factorial(0) is 1
  - factorial(1) is 1 (1 * factorial(0))
  - factorial(2) is 2 (2 * factorial(1))
  - factorial(3) is 6 (3 * factorial(2))
  - factorial(4) is 24 (4 * factorial(3))

- factorial(n) is n * factorial(n – 1)
public static int factorial(int num) {
    if (num < 0) {
        // Factorial is not defined for values less than zero, let -1 indicate error
        return -1;
    } else if (num == 0) {
        // factorial(0) is 1
        return 1;
    } else { // (num > 0)
        // factorial(n) is n * factorial(n - 1)
        return num * factorial(num - 1);
    }
}
Exponent

- Exponent can also be expressed recursively
  - \( \text{pow}(\text{base}, 0) = 1 \)
  - \( \text{pow}(\text{base}, 1) = \text{base} \times 1 \)
  - \( \text{pow}(\text{base}, 2) = \text{base} \times \text{base} \times 1 \)
Exponent

- Exponent can also be expressed recursively
  - $\text{pow}(\text{base}, 0) = 1$
  - $\text{pow}(\text{base}, 1) = \text{base} \times \text{pow}(\text{base}, 0)$
  - $\text{pow}(\text{base}, 2) = \text{base} \times \text{pow}(\text{base}, 1)$
  - $\text{pow}(\text{base}, \text{exp}) = \text{base} \times \text{pow}(\text{base}, \text{exp} - 1)$
Recursive Exponent

```java
public static double pow(double base, int power)
{
    if (power == 0)
    {
        // base ^ 0 is 1, lets ignore 0 ^ 0 or 0 ^ -power
        // for now
        return 1;
    }
    else if (power > 0)
    {
        // Positive exponent
        return pow(base, power - 1) * base;
    }
    else // (power < 0)
    {
        // Negative exponent
        return pow(base, power + 1) / base;
    }
}
```
Iterative v. Recursive Methods

- We’ve seen exponent implemented without recursion
  - It used a loop
public static int factorial(int num) {
    if (num < 0) {
        // factorial is not defined for negative values, so return -1 as error
        return -1;
    }

    int product = 1;
    for (int i = 2; i <= num; i++) {
        product *= i;
    }

    return product;
}

public static int factorial(int num) {
    if (num < 0) {
        // Factorial is not defined for values less than zero, let -1 indicate error
        return -1;
    }

    else if (num == 0) {
        // factorial(0) is 1
        return 1;
    }

    else { // (num > 0)
        // factorial(n) is n * factorial(n - 1)
        return num * factorial(num - 1);
    }
}
Exponential

**Iterative**

```java
public static double pow(
    double base, int power)
{
    double exponent = 1.0;
    if (power < 0)
    {
        for (int i = 0; i < -power; i++)
        {
            exponent /= base;
        }
    }
    else if (power > 0)
    {
        for (int i = 0; i < power; i++)
        {
            exponent *= base;
        }
    }
    return exponent;
}
```

**Recursive**

```java
public static double pow(
    double base, int power)
{
    if (power == 0)
    {
        // base ^ 0 is 1
        return 1;
    }
    else if (power > 0)
    {
        // Positive exponent
        return pow(base, power - 1) * base;
    }
    else // (power < 0)
    {
        // Negative exponent
        return pow(base, power + 1) / base;
    }
}
```
Iterative v. Recursive Methods

- We’ve seen exponent implemented without recursion
  - It used a loop

- All recursive methods can be expressed as iterative methods
  - Typically a loop fills in for the recursive call
  - Sometimes it is clearer to write an algorithm using recursion
Merge Sort

- Merge sort is a more time-efficient method of sorting an array than selection sort or bubble sort

1. If array has only 1 element, then return
2. Copy the first half of array into a smaller array named firstHalf
3. Copy the rest of the array into a smaller array named lastHalf
4. Sort firstHalf using a recursive call
5. Sort lastHalf using a recursive call
6. Merge the elements in firstHalf and lastHalf into array
Merge Sort: sort

```java
public static void sort(int[] array) {
    if (array.length >= 2) {
        int halfLength = array.length / 2;
        int[] firstHalf = new int[halfLength];
        int[] lastHalf =
            new int[array.length - halfLength];
        divide(array, firstHalf, lastHalf);
        sort(firstHalf);
        sort(lastHalf);
        merge(array, firstHalf, lastHalf);
    } // else do nothing since an array of 0 or 1
    // elements is already sorted
}
```

Recursive Calls

`divide(array, firstHalf, lastHalf);`

`sort(firstHalf);`

`sort(lastHalf);`

`merge(array, firstHalf, lastHalf);`
private static void divide(
    int[] array, int[] firstHalf, int[] lastHalf)
{
    // Copy the first half of the elements from
    // array into firstHalf
    for (int i = 0; i < firstHalf.length; i++)
    {
        firstHalf[i] = array[i];
    }

    // Copy the second half of the elements from
    // array into lastHalf
    for (int i = 0; i < lastHalf.length; i++)
    {
        lastHalf[i] = array[firstHalf.length + i];
    }
}
Merge Sort: merge (part 1)

```java
private static void merge(int[] array, int[] firstHalf, int[] lastHalf) {
    int firstHalfIndex = 0, lastHalfIndex = 0, arrayIndex = 0;

    while ((firstHalfIndex < firstHalf.length) && (lastHalfIndex < lastHalf.length)) {
        if (firstHalf[firstHalfIndex] < lastHalf[lastHalfIndex]) {
            // The element in the first half is smaller than that of the
            // last half, so copy the first half's element into the result
            array[arrayIndex] = firstHalf[firstHalfIndex];
            firstHalfIndex++;
        } else {
            // The current element in the last half is smaller than that of
            // the first half, so copy the last half's element into the
            // result
            array[arrayIndex] = lastHalf[lastHalfIndex];
            lastHalfIndex++;
        }
        arrayIndex++;
    }
}
```
Merge Sort: merge (part 2)

// If there are still elements left in firstHalf, then
// they are bigger than anything in lastHalf so copy them
// to the result
while (firstHalfIndex < firstHalf.length)
{
    array[arrayIndex] = firstHalf[firstHalfIndex];
    arrayIndex++;
    firstHalfIndex++;
}

// If there are still elements left in lastHalf, then
// they are bigger than anything in lastHalf so copy them
// to the result
while (lastHalfIndex < lastHalf.length)
{
    array[arrayIndex] = lastHalf[lastHalfIndex];
    arrayIndex++;
    lastHalfIndex++;
}
Questions?
Logistics

Next:

- Lab 8