Programming in Java: lecture 10

- Recursion
  - Why?
  - How?
  - Examples
- Example

Slides made for use with ”Introduction to Programming Using Java, Version 5.0” by David J. Eck
Some figures are taken from ”Introduction to Programming Using Java, Version 5.0” by David J. Eck
Lecture 10 covers Section 9.1

Recursion

- Defining something by itself
  - usually a bad idea
- Defining something partially by itself
  - a very powerful technique
- Calling a method from itself
  - can be done indirectly

Recursion

- Base cases
- Splitting the problem into smaller problems
- The problem of infinite recursion

Binary Search

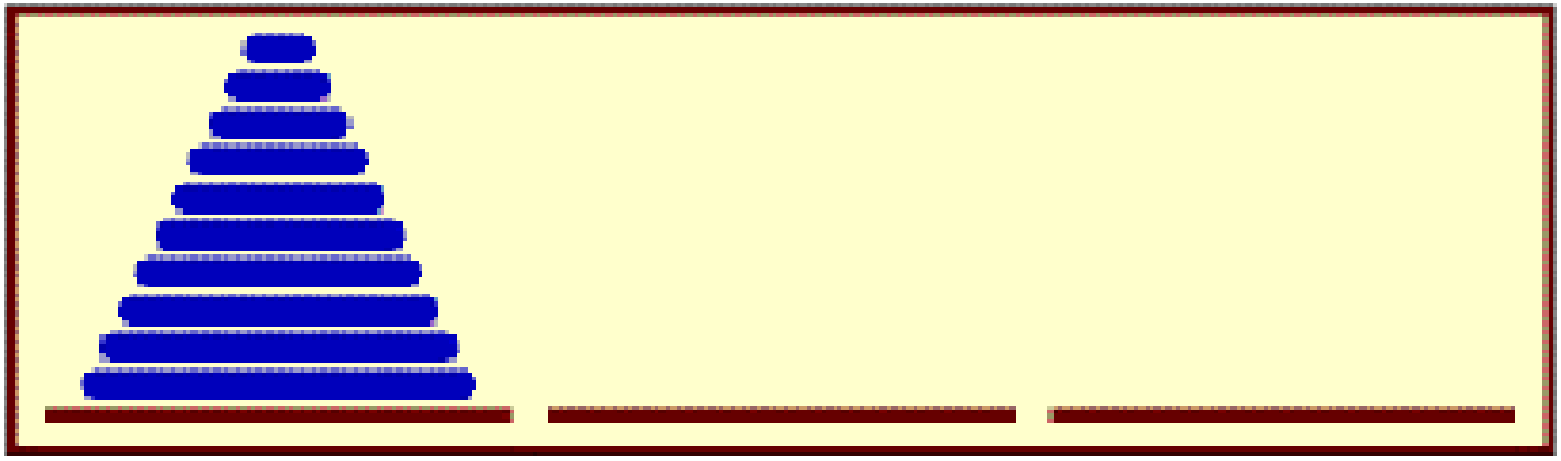
- Why?
- Liniear search
  - 1000 items, max 1000 comparisons
  - 1000000 items, max 1000000 comparisons
- Binary search
  - 1000 items, 10 comparisons
  - 1000000 items, 20 comparisons
  - Data must be sorted

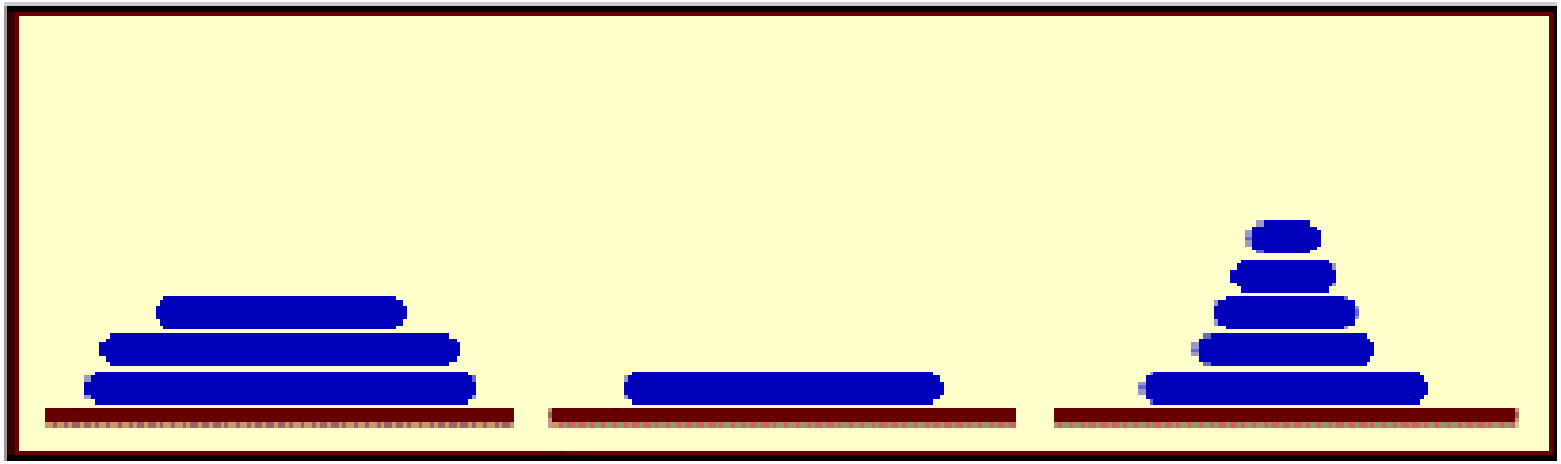
```
 * Searches the array A for the integer N.
 * Precondition:  A must be sorted into increasing order.
 * Postcondition: If N is in the array, then the return value, i,
 *    satisfies A[i] == N.  If N is not in the array, then the
 *    return value is -1.
 */
static int binarySearch(int[] A, int N) {

    int lowestPossibleLoc = 0;
    int highestPossibleLoc = A.length - 1;

    while (highestPossibleLoc >= lowestPossibleLoc) {
       int middle = (lowestPossibleLoc + highestPossibleLoc) / 2;
       if (A[middle] == N) {
                  // N has been found at this index!
          return middle;
       }
       else if (A[middle] > N) {
                  // eliminate locations >= middle
          highestPossibleLoc = middle - 1;
       }
       else {
                  // eliminate locations <= middle
          lowestPossibleLoc = middle + 1;
       }
    }

    // At this point, highestPossibleLoc < LowestPossibleLoc,
    // which means that N is known to be not in the array.  Return
    // a -1 to indicate that N could not be found in the array.

    return -1;

}
```

Recursive Binary Search

```
static int binarySearch(int[] A, int loIndex, int hiIndex, int value) {

   if (loIndex > hiIndex) {
         // The starting position comes after the final index,
         // so there are actually no elements in the specified
         // range.  The value does not occur in this empty list!
      return -1;
   }

   else {
         // Look at the middle position in the list.  If the
         // value occurs at that position, return that position.
         // Otherwise, search recursively in either the first
         // half or the second half of the list.
      int middle = (loIndex + hiIndex) / 2;
      if (value == A[middle])
         return middle;
      else if (value < A[middle])
         return binarySearch(A, loIndex, middle - 1, value);
      else   // value must be > A[middle]
         return binarySearch(A, middle + 1, hiIndex, value);
   }

} // end binarySearch()
```

Towers of Hanoi





The stacks after a number of moves.

Algorithm

```
/**
 * Solve the problem of moving the number of disks specified
 * by the first parameter from the stack specified by the
 * second parameter to the stack specified by the third
 * parameter.  The stack specified by the fourth parameter
 * is available for use as a spare.  Stacks are specified by
 * number: 0, 1, or 2.
 */
static void TowersOfHanoi(int disks, int from, int to, int spare) {
   if (disks == 1) {
         // There is only one disk to be moved.  Just move it.
      System.out.println("Move a disk from stack number "
               + from + " to stack number " + to);
   }
   else {
         // Move all but one disk to the spare stack, then
         // move the bottom disk, then put all the other
         // disks on top of it.
      TowersOfHanoi(disks-1, from, spare, to);
      System.out.println("Move a disk from stack number "
               + from + " to stack number " + to);
      TowersOfHanoi(disks-1, spare, to, from);
   }
}
```

Solution

```
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 1 to stack number 0
Move a disk from stack number 1 to stack number 2
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 2 to stack number 0
Move a disk from stack number 1 to stack number 0
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
```

Quicksort

To apply QuicksortStep to a list of numbers, select one of the numbers, 23 in this case. Arrange the numbers so that numbers less than 23 lie to its left and numbers greater than 23 lie to its right.

23 10 7 45 16 86 56 2 31 18

18 12 7 2 16 23 86 56 31 45

To finish sorting the list, sort the numbers to the left of 23, and sort the numbers to the right of 23. The nuber 23 itself is already in its final position and doesn't have to be moved again

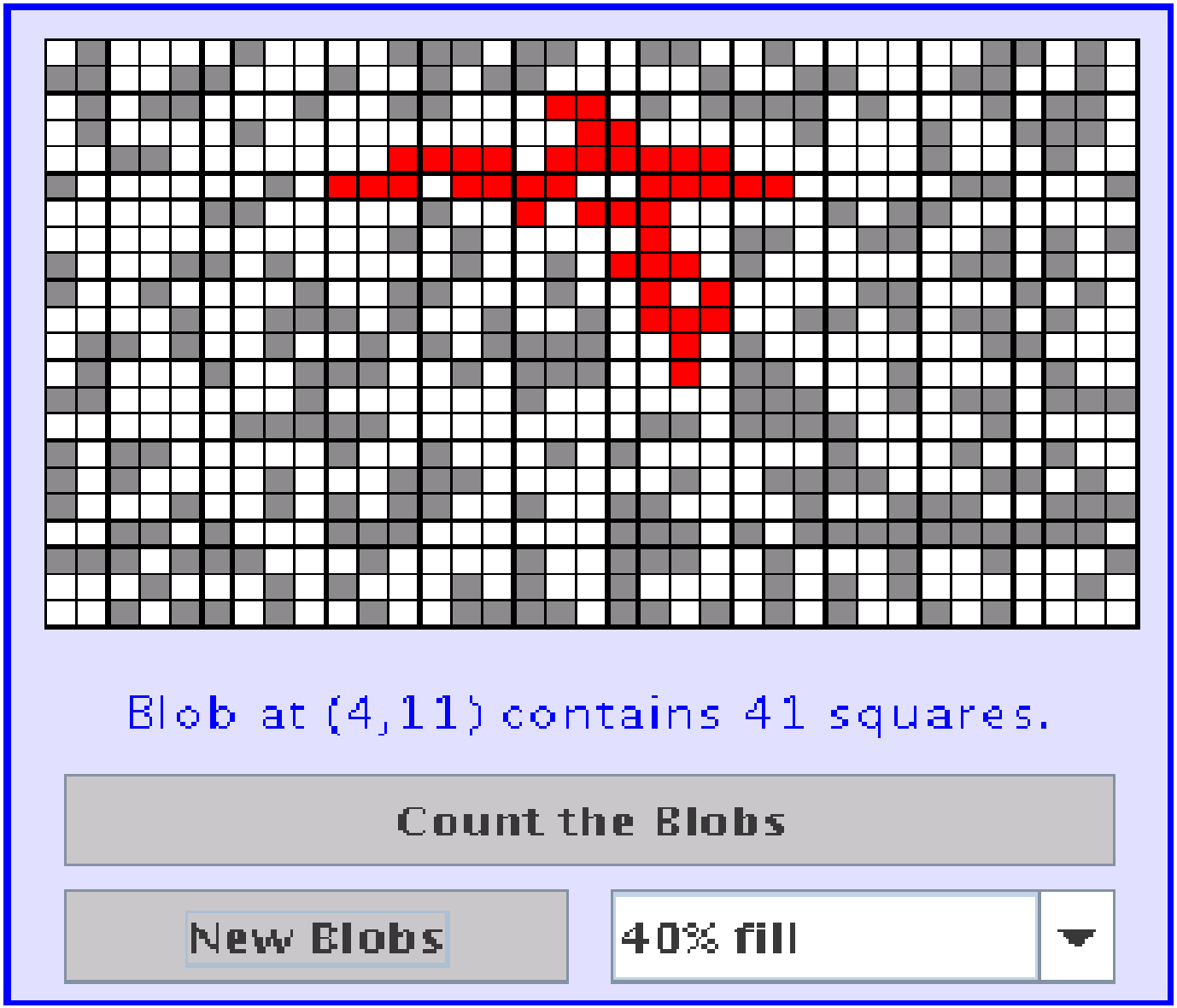
Quicksort

```
/**
 * Apply quicksort to put the array elements between
 * position lo and position hi into increasing order.
 */
static void quicksort(int[] A, int lo, int hi) {
   if (hi <= lo) {
         // The list has length one or zero.  Nothing needs
         // to be done, so just return from the subroutine.
      return;
   }
   else {
         // Apply quicksortStep and get the new pivot position.
         // Then apply quicksort to sort the items that
         // precede the pivot and the items that follow it.
      int pivotPosition = quicksortStep(A, lo, hi);
      quicksort(A, lo, pivotPosition - 1);
      quicksort(A, pivotPosition + 1, hi);
   }
}
```

Quicksort step

```
static int quicksortStep(int[] A, int lo, int hi) {

   int pivot = A[lo];  // Get the pivot value.

   while (hi > lo) {
      while (hi > lo && A[hi] > pivot) {
         hi--;
      }
      if (hi == lo)
         break;
      A[lo] = A[hi];
      lo++;
      while (hi > lo && A[lo] < pivot) {
         lo++;
      }
      if (hi == lo)
         break;
      A[hi] = A[lo];
      hi--;
   } // end while
   A[lo] = pivot;
   return lo;
}  // end QuicksortStep
```

Blob counting



```
int getBlobSize(int r, int c) {  // BUGGY, INCORRECT VERSION!!
      // This INCORRECT method tries to count all the filled
      // squares that can be reached from position (r,c) in the grid.
   if (r < 0 || r >= rows || c < 0 || c >= columns) {
         // This position is not in the grid, so there is
         // no blob at this position.  Return a blob size of zero.
      return 0;
   }
   if (filled[r][c] == false) {
        // This square is not part of a blob, so return zero.
      return 0;
   }
   int size = 1;  // Count the square at this position, then count th
                  //   the blobs that are connected to this square
                  //   horizontally or vertically.
   size += getBlobSize(r-1,c);
   size += getBlobSize(r+1,c);
   size += getBlobSize(r,c-1);
   size += getBlobSize(r,c+1);
   return size;
}  // end INCORRECT getBlobSize()
```

Infinite recursion

- StackOverflowError

```
int getBlobSize(int r, int c) {
   if (r < 0 || r >= rows || c < 0 || c >= columns) {
         // This position is not in the grid, so there is
         // no blob at this position.  Return a blob size of zero.
      return 0;
   }
   if (filled[r][c] == false || visited[r][c] == true) {
        // This square is not part of a blob, or else it has
        // already been counted, so return zero.
      return 0;
   }
   visited[r][c] = true;    // Mark the square as visited so that
                            //    we won't count it again during the
                            //    following recursive calls.
   int size = 1;  // Count the square at this position, then count the
                  //   the blobs that are connected to this square
                  //   horizontally or vertically.
   size += getBlobSize(r-1,c);
   size += getBlobSize(r+1,c);
   size += getBlobSize(r,c-1);
   size += getBlobSize(r,c+1);
   return size;
}  // end getBlobSize()
```

```
void countBlobs() {

   int count = 0; // Number of blobs.

   /* First clear out the visited array. The getBlobSize() method
      will mark every filled square that it finds by setting the
      corresponding element of the array to true.  Once a square
      has been marked as visited, it will stay marked until all the
      blobs have been counted.  This will prevent the same blob from
      being counted more than once. */

   for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++)
         visited[r][c] = false;

   /* For each position in the grid, call getBlobSize() to get the
      size of the blob at that position.  If the size is not zero,
      count a blob.  Note that if we come to a position that was part
      of a previously counted blob, getBlobSize() will return 0 and
      the blob will not be counted again. */

   for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++) {
         if (getBlobSize(r,c) > 0)
            count++;
      }

   repaint();  // Note that all the filled squares will be red,
               //   since they have all now been visited.

   message.setText("The number of blobs is " + count);

} // end countBlobs()
```

Example

- Team programming