Unit 10: Recursion

Adapted from:

1) Building Java Programs: A Back to Basics Approach
by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum

https://longbaonguyen.github.io

Recursion

recursion: The definition of an operation in terms of itself.

- Solving a problem using recursion depends on solving smaller occurrences of the same problem.

recursive programming: Writing methods that call themselves to solve problems recursively.

- An equally powerful substitute for *iteration* (loops)
- Particularly well-suited to solving certain types of problems

Why learn recursion?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **base case**: A simple occurrence that can be answered directly.
  - **recursive case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
  - Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
  - A crucial part of recursive programming is identifying these cases.

Example

You are lined up in front of your favorite store for Black Friday deals. The line is long and wraps around the building so that you cannot see the front of the line. How do you figure out your position without getting out of line?

Answer: Ask the person in front of you.

Base case: If a customer is at the front of the line and someone asks him for his position, he'll "return" 1.

Recursive case: If a customer is at position n and someone asks him for his position, he'll ask the person in front of him.

Note: The recursive case reduces an n problem to an n-1 problem. Each person repeated asks until the question reaches the first person in line. He will answer informing the person behind him, who will then inform the person behind him, etc… until the answer reaches you.

Recursion in Java

- Consider the following method to print a line of * characters:

```
// Prints a line containing the given number of stars.
// Precondition: n >= 0
public static void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();   // end the line of output
}
```

- Write a recursive version of this method (that calls itself).
  - Solve the problem without using any loops.
  - Hint: Your solution should print just one star at a time.

"Recursion Zen"

The real, even simpler, base case is an `n` of 0, not 1:

```
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    }
    else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

- **Recursion Zen**: The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

Exercise

Write a recursive method `pow` accepts an integer base and exponent and returns the base raised to that exponent.

- Example: `pow(3, 4)` returns 81
- Solve the problem recursively and without using loops.

```
// Precondition: exponent >= 0, base > 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    }
    else {
        // recursive case:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

Recursive Trace 1

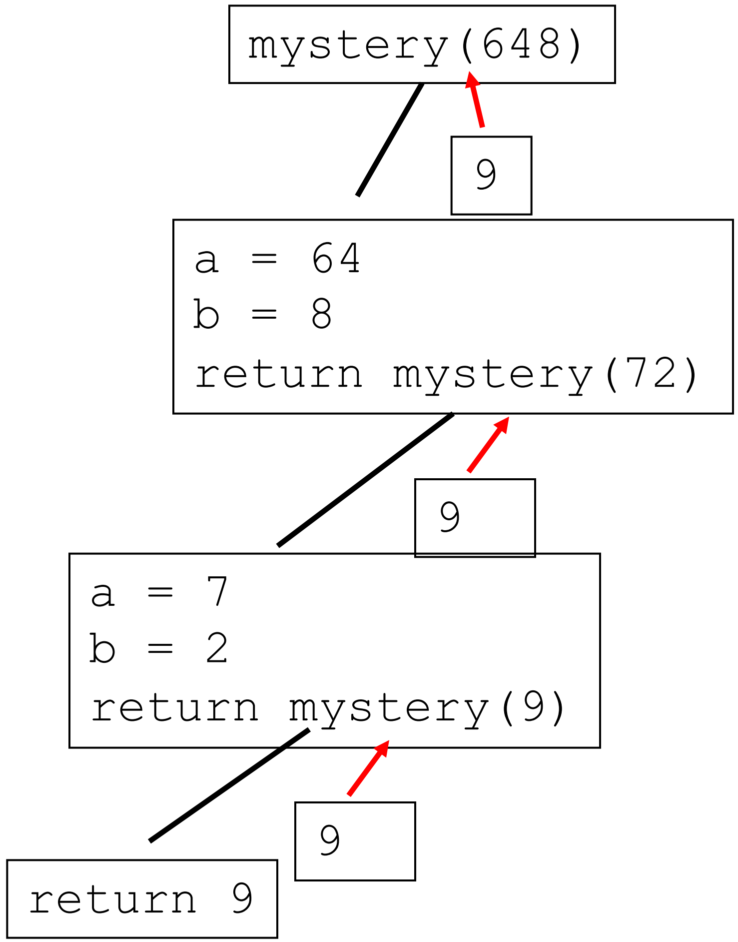
- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

  - What is the result of the following call?

    `mystery(648)`

Recursion Tree Diagram 1



9 propagates all the up to mystery(648) which equals 9.

Note: This is the simplest example where the same number progagates up. Usually, at each step, more math is performed on each answer.

Recursive Trace 2

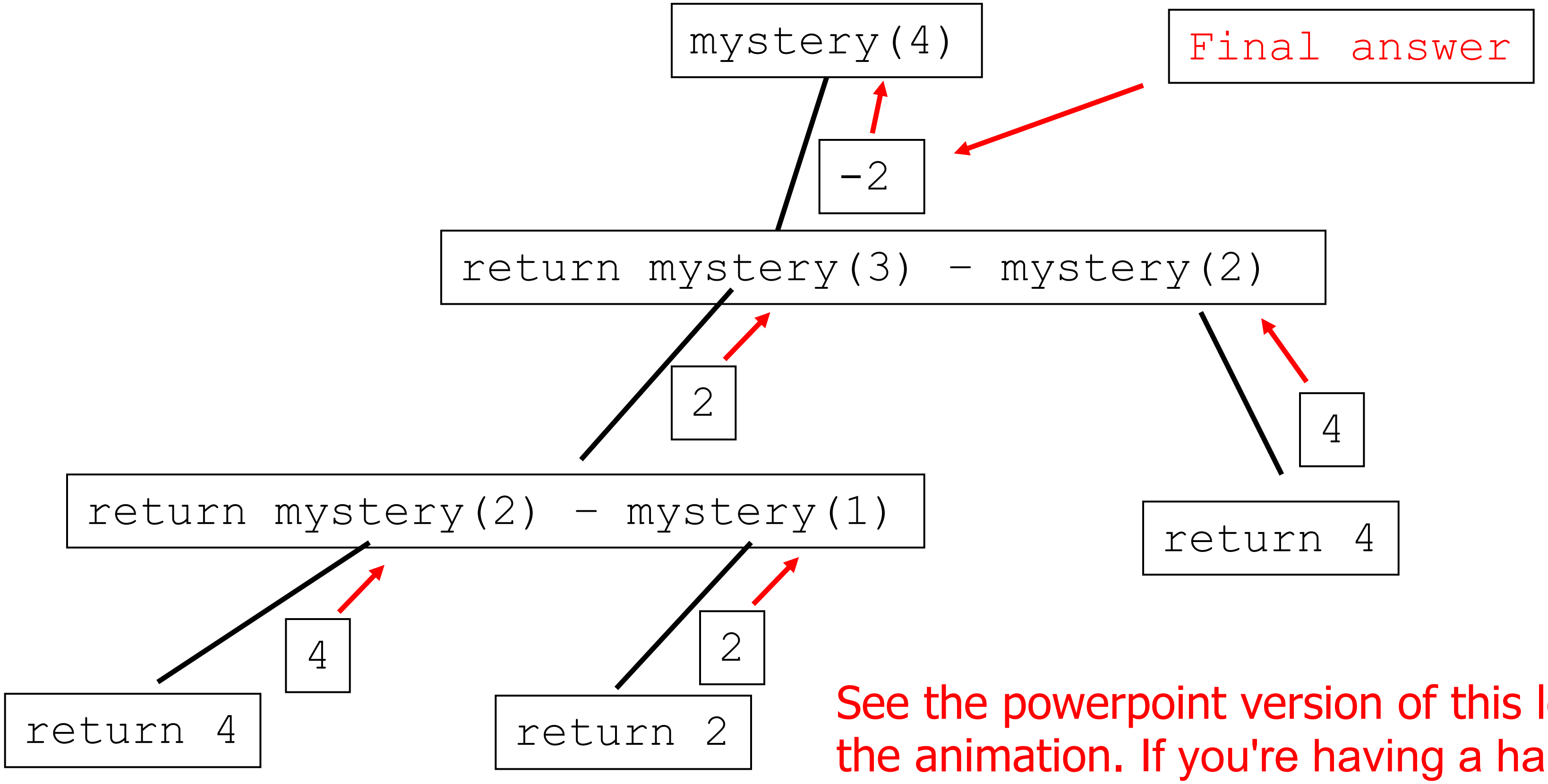
```
int mystery(int n){
   if (n == 1 || n == 2)
      return 2 * n;
   else
      return mystery(n - 1) - mystery(n - 2);
}
```

What is the result of the following call?

```
mystery(4);
```

See the next slide for a way to visualize this one. Watch the animation on the powerpoint version of this lecture.

Recursion Tree Diagram 2



See the powerpoint version of this lecture for the animation. If you're having a hard time visualizing recursion, I highly recommend watching the animation of this slide.

Recursive Trace 3

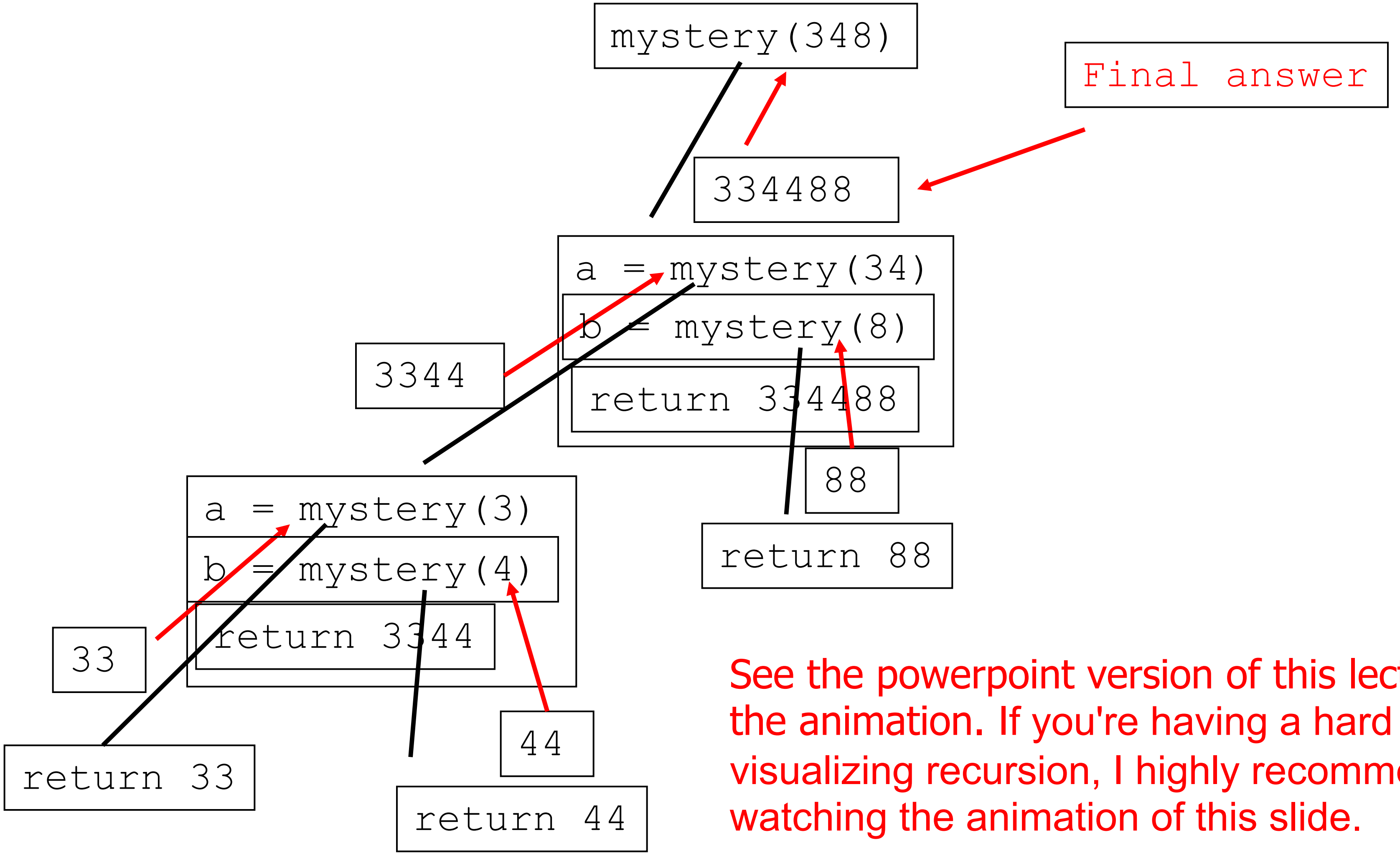
- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

  - What is the result of the following call?

    `mystery(348)`

Recursion Tree Diagram 3



See the powerpoint version of this lecture for the animation. If you're having a hard time visualizing recursion, I highly recommend watching the animation of this slide.

Recursive Binary Search

The following implementation of recursive binary search has appeared on the AP Exam. Understand this algorithm!

```
public static int bSearch(int[] arr, int left,
                                              int right, int x){
    if (right >= left){
        int mid = (left + right) / 2;
        if (arr[mid] == x){
          return mid;
        }
        else if (arr[mid] > x){
          return bSearch(arr, left, mid - 1, x);
        }
        else{
          return bSearch(arr, mid + 1, right, x);
      }
    }
  return -1;
}
```

Merge sort

merge sort: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

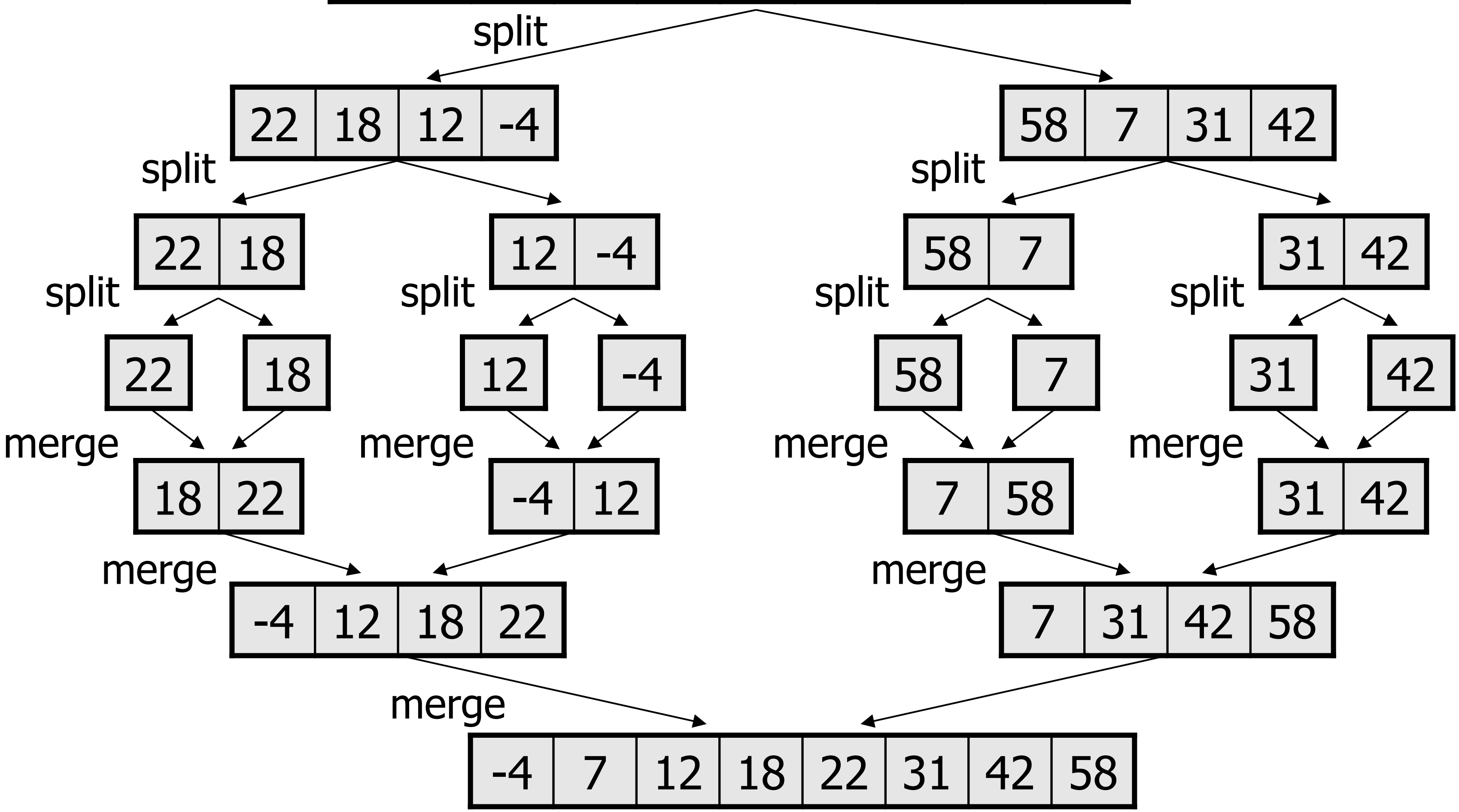
The algorithm:

- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945

Merge sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



Merging sorted halves

Subarrays (left) 0	1	2	3	Subarrays (right) 0	1	2	3	Next include	Merged array 0	1	2	3	4	5	6	7
14 (i1)	32	67	76	**23** (i2)	41	58	85	14 from left	14 (i)							
14	**32** (i1)	67	76	**23** (i2)	41	58	85	23 from right	14	23 (i)						
14	**32** (i1)	67	76	23	**41** (i2)	58	85	32 from left	14	23	32 (i)					
14	32	**67** (i1)	76	23	**41** (i2)	58	85	41 from right	14	23	32	41 (i)				
14	32	**67** (i1)	76	23	41	**58** (i2)	85	58 from right	14	23	32	41	58 (i)			
14	32	**67** (i1)	76	23	41	58	**85** (i2)	67 from left	14	23	32	41	58	67 (i)		
14	32	67	**76** (i1)	23	41	58	**85** (i2)	76 from left	14	23	32	41	58	67	76 (i)	
14	32	67	76	23	41	58	**85** (i2)	85 from right	14	23	32	41	58	67	76	85 (i)

Merge halves code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                                       int[] right) {
    int i1 = 0;    // index into left array
    int i2 = 0;    // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
           (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];   // take from right
            i2++;
        }
    }
}
```

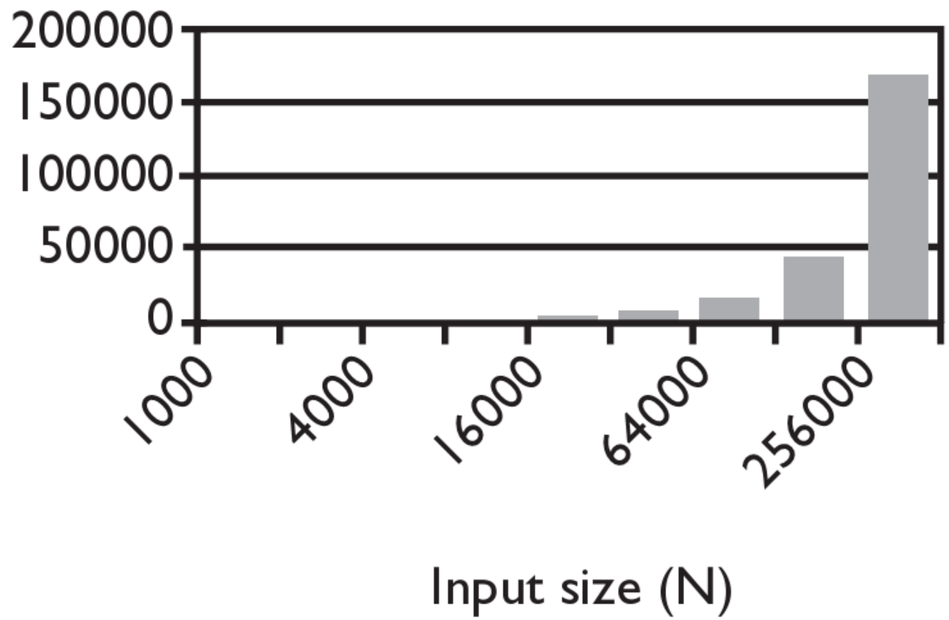
Merge sort code

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

Selection sort runtime (Fig. 13.6)

What is the complexity class (Big-Oh) of selection sort?

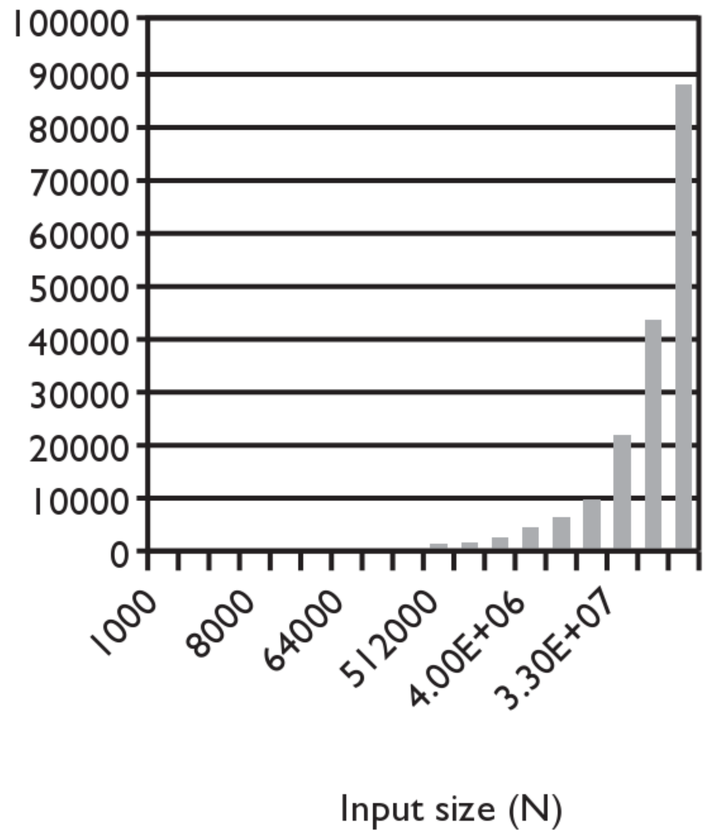
N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Merge sort runtime

- What is the complexity class (Big-Oh) of merge sort?

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



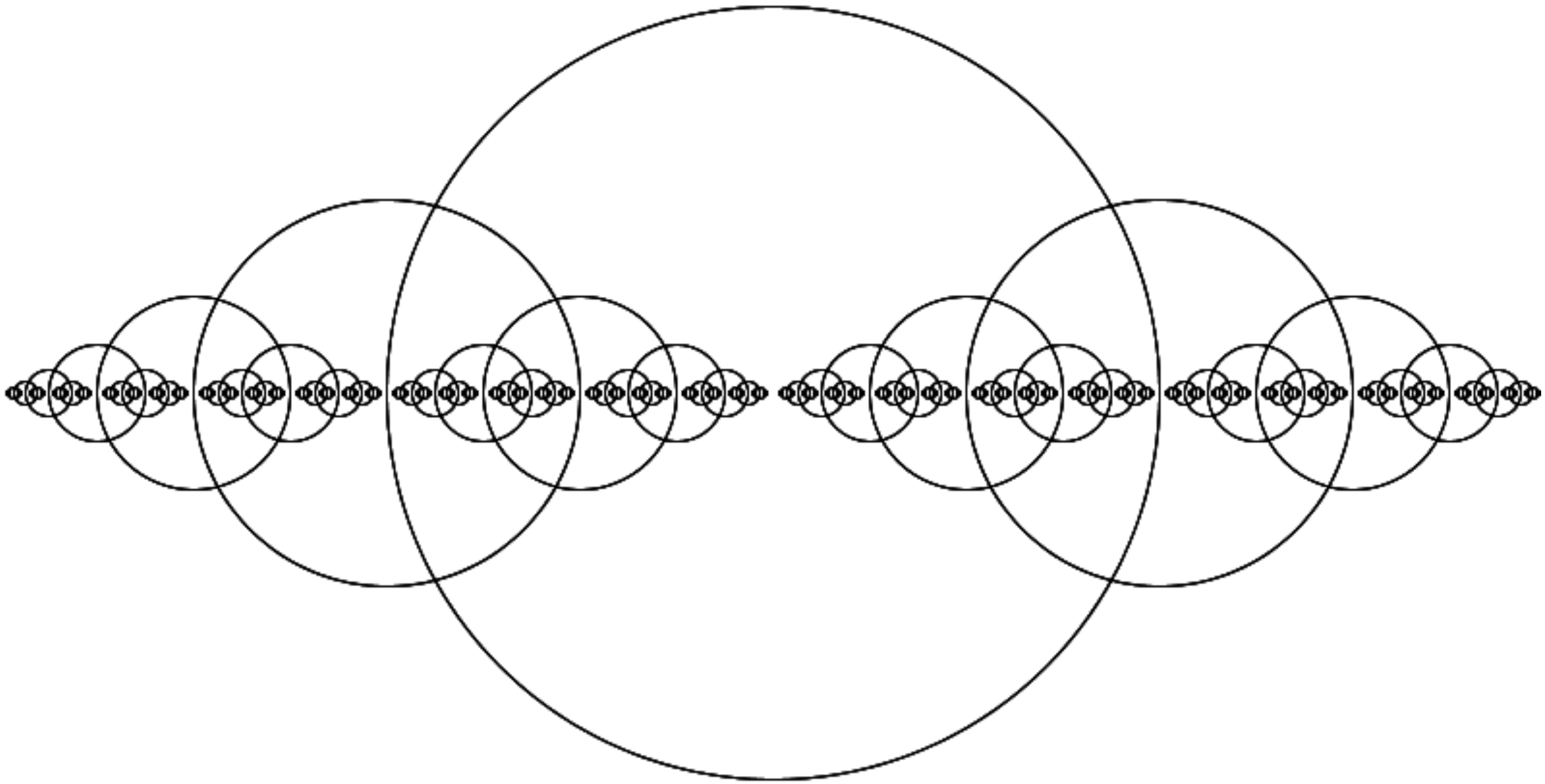
Lab 1: Fractal Circles

Use Processing to write the recursive method to print out a recursive pattern of circles of decreasing radii. Since this is a static image, you don't need draw(). Use the following complete template:

```
void setup() {
   background(255);
   size(600,600);
   circle(width/2,width/4,10);
}
void circle(int x, int radius, int depth)
{
  // fill in your code
  //(just a 4 lines of code inside an if conditional!)


}
```

Lab 1

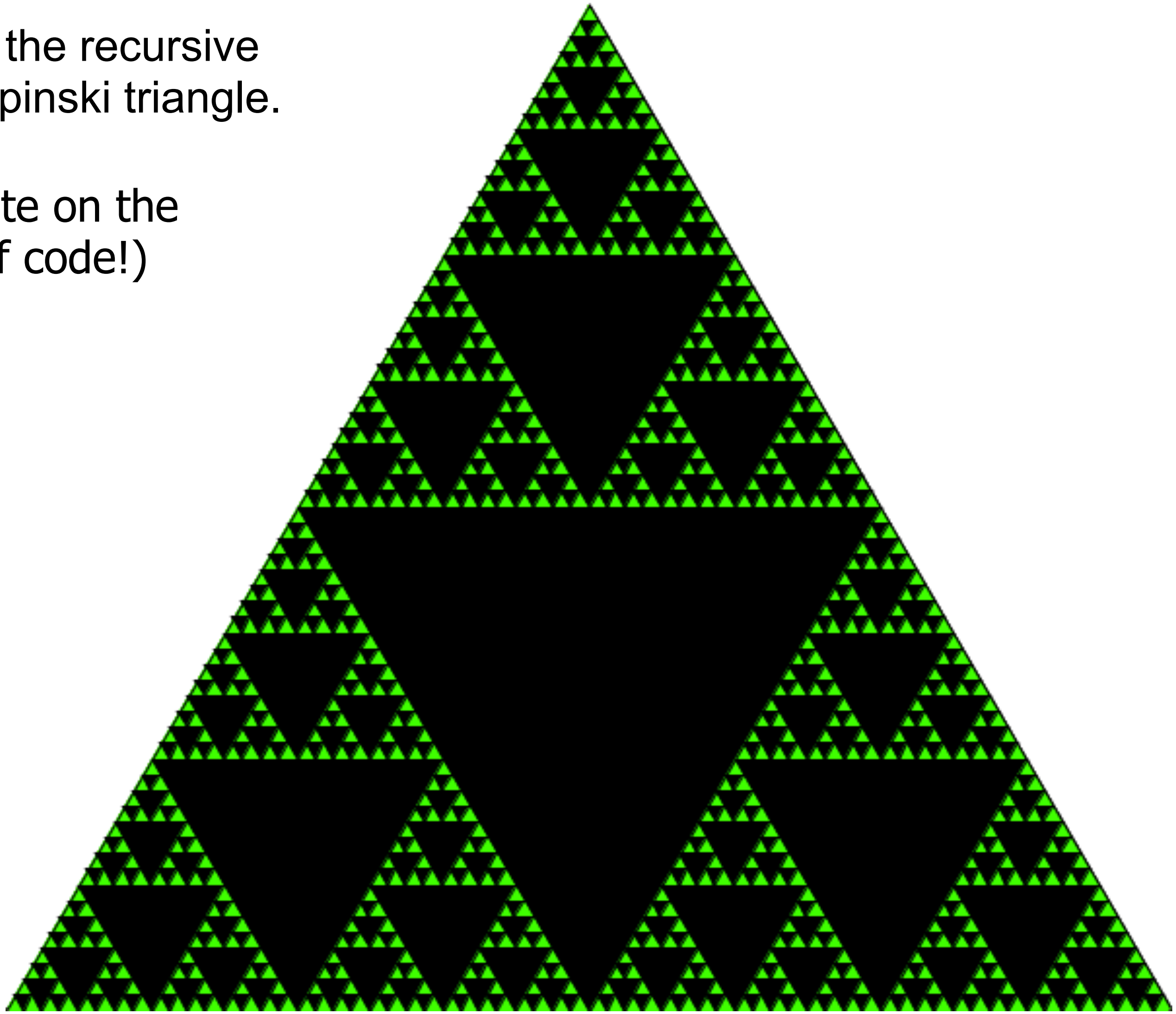
The previous template with correct completed code produces:



Lab 2: Sierpinski Triangle

Use Processing to write the recursive method to draw the Sierpinski triangle.

You may use the template on the next slide.(just 5 lines of code!)



Lab 2: Sierpinski Triangle

```
void setup() {
   background(255);
   size(600,600);
   fill(0,255,0);
   // draw the first green triangle.
   triangle(0,height, width/2, 0, width, height);

   // start the recursive call.
   fractal(0,height, width/2, 0, width, height, 6);
}

void fractal(int x1, int y1, int x2, int y2, int x3,
                                  int y3, int n) {

  // fill in your code
  //(just a 5 (algebraically careful) lines of code
  // inside an if conditional!)

}
```