# Recursion — Pt 2

# Recursion — returning values & private recursive functions

```
private long fibonacciUtil(long fibNumA, longfibNumB, int counter)
    {
        System.out.println(counter + " " + fibNumA + " " + fibNumB);
        if (counter==1)
            return fibNumA.add(fibNumB);
        return fibonacciUtil(fibNumB, fibNumA + fibNumB, counter-1);
    }

   public long fibonacci(int n) {
        if (n<=0) return 0;
        if (n<3)  return 1;
        return fibonacciUtil(1, 1, n-2);
    }
```

See also BadFibb.java

# Returning an ArrayList

Make a method that returns an array list containing N numbers, starting at M then 2M, 3M, 4M, ...

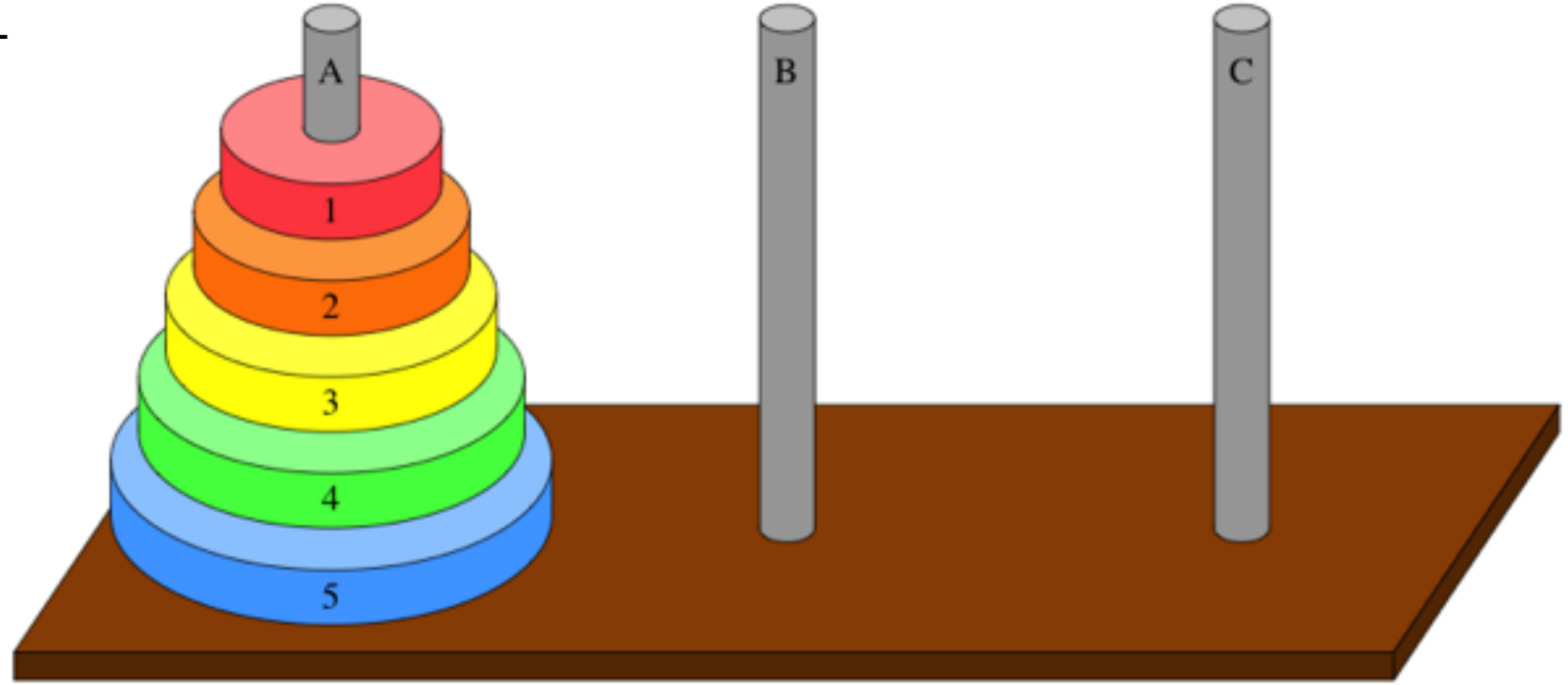
# Edouard Lucas

- French Mathematician
  - 1842 - 1891
- proved 2^127-1 is prime
  - largest known  Mersenne prime for 75 years

- “lucas sequences” simplest of these is the Fibonacci sequence

- Invented/published the “dots and boxes” game
- The ~~Towers of Hanoi~~  Lucas Towers

# Lucas Towers




Extra Credit: +10 to homework
Must be done by last day of classes
Do in lab with a TA witness

# Count the number of occurrences of a letter in a string

```
public int numOccur1(char ch, String str) {
     if (str == null || str.equals("")) {
         return 0;
     }
     int count = 0;
     if (str.charAt(0) == ch) {
         count++;
     }
     numOccur1(ch, str.substring(1));
     return count;
 }
```

What does this return on “a” , “abc”
Why?

# Occurrence count v2

```
int acount = 0;

 public int numOccur2(char ch, String str) {
     if (str == null || str.equals("")) {
         return 0;
     }
     if (str.charAt(0) == ch) {
         acount++;
     }
     numOccur2(ch, str.substring(1));
     return acount;
 }
```

Correct answer, but a BAD solution

# Occurrence count v3 and v4

```
public int numOccur3(char ch, String str) {
    if (str == null || str.equals("")) { return 0; }
    int count = 0;
    if (str.charAt(0) == ch) { count = 1; }
    return count + numOccur3(ch, str.substring(1));
}


public int numOccur4(char ch, String str) {
    return numOccur4Util(ch, str, 0);
}
private int numOccur4Util(char ch, String str, int count) {
    if (str == null || str.equals("")) { return count; }
    if (str.charAt(0) == ch) { count++; }
    return numOccur4Util(ch, str.substring(1), count);
}
```

# v5 and v6

```
public int numOccur5(char ch, String str) {
        if (str == null || str.length()==0)
            return 0;
        return numOccur5Util(ch, str, 0, 0);
    }
private int numOccur5Util(char ch, String str, int loc, int count) {
        if (loc >= str.length())
            return count;
        if (str.charAt(loc) == ch) { count++; }
        return numOccur5Util(ch, str, loc+1, count);
    }

public int numOccur6(char ch, String str) {
        if (str == null || str.length()==0)
            return 0;
        return numOccur6Util(ch, str, 0);
    }
private int numOccur6Util(char ch, String str, int loc) {
        if (loc >= str.length())
            return 0;
        return (str.charAt(loc)==ch ? 1 : 0) + numOccur6Util(ch, str, loc+1);
   }
```

Ternary Operator!

# Practice

- Recursively compute the number of odd numbers in an array of integers
- Recursively count the number of strings of at least a given length in an ArrayList of strings
  - for instance, given a length of 0, this would return the count of the non-null strings in the array

# Finding a data item

- Suppose you have an array (or ArrayList) of N items.  How do you determine if the array contains a particular item?
  - Does the form of the array matter?
    - Unsorted
    - Sorted
  - What is the complexity of finding an item?

# Naive Find

## works on all arrays. Sorted or Unsorted

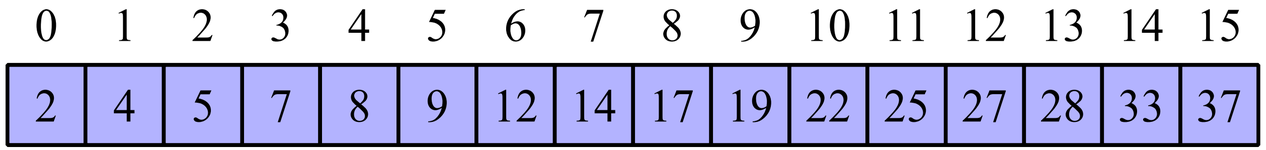
- Start at beginning
  - compare until found
  - Time Complexity??
  - Loops would work

```
public int find(int[] arr, int num) {
        return findUtil(arr, num, 0);
   }

    /**
     * Find be looking at each item.  The array may be in any order
     * @param arr the array to be searched
     * @param num the number to be found
     * @param loc the location to consider next
     * @return the location of num in arr, or -1 if not in arr
     */
    private int findUtil(int[] arr, int num, int loc) {
        if (loc >= arr.length)
            return -1;
        if (arr[loc] == num)
            return loc;
        return findUtil(arr, num, loc + 1);
    }
```

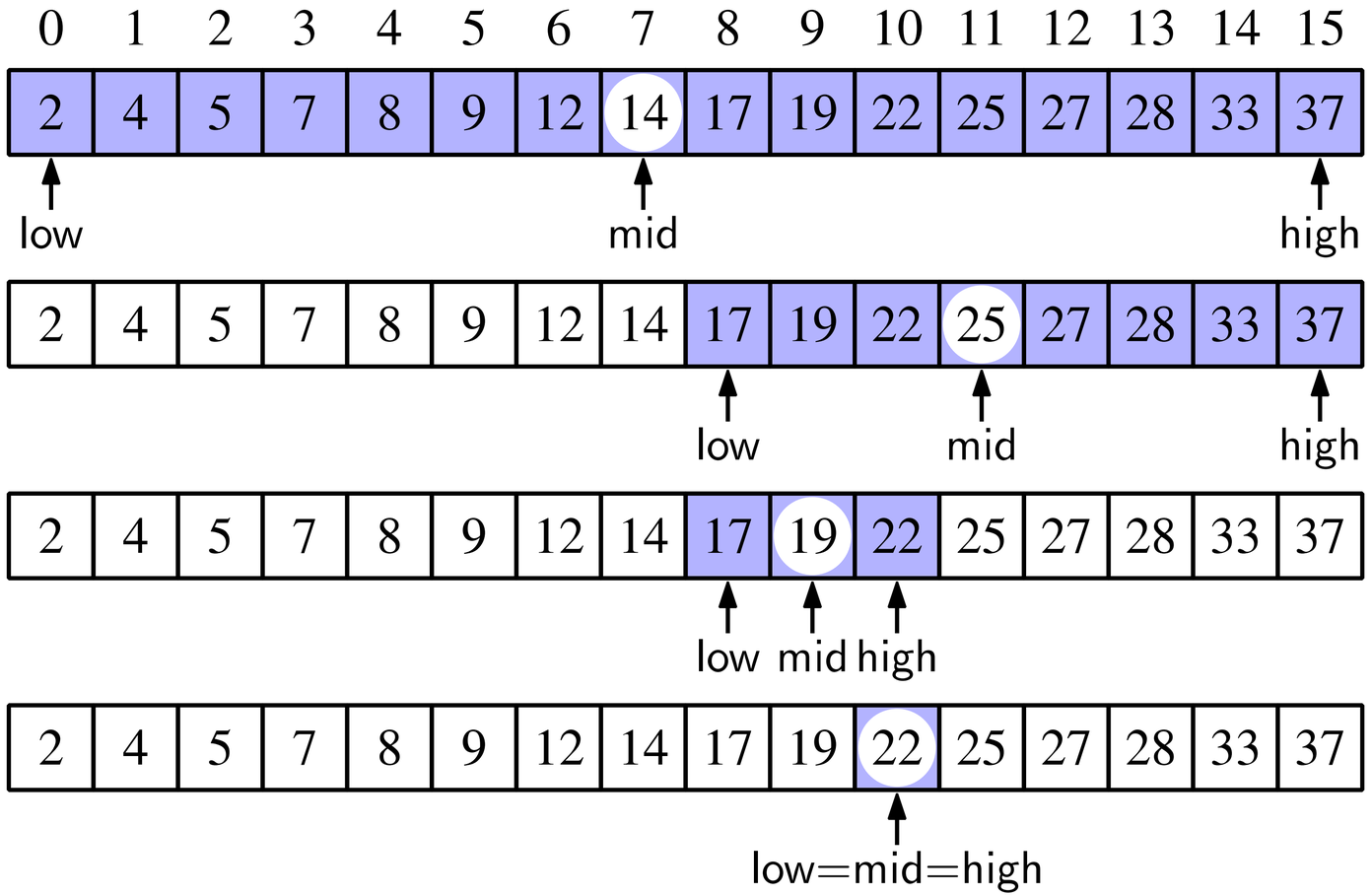
# Binary Search
# Faster find on sorted arrays

- Search for an integer (22) in an ordered list




- $mid = \left\lfloor \frac{low + high}{2} \right\rfloor = \left\lfloor \frac{0 + 15}{2} \right\rfloor = 7$
  - `target==data[mid]`, found
  - `target>data[mid]`, recur on second half
  - `target<data[mid]`, recur on first half

# target = 22


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low mid high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low=mid=high

# Binary Search Code

```
/**
     * An alternate version of finding number in an array
     * This version requires that the array be sorted
     * @param arr the array - it must be sorted
     * @param num the number to be found
     * @param minPos the lowest position that the item to be found could be at
     * @param maxPos the highest possible location
     * @return the location, or -1 if not found
     */
    private int findUtilB(int[] arr, int num, int minPos, int maxPos) {
        if (minPos >= maxPos)
            return -1;
        int loc = (minPos + maxPos) / 2;
        if (arr[loc] == num)
            return loc;
        if (num > arr[loc])
            return findUtilB(arr, num, loc + 1, maxPos);
        else
            return findUtilB(arr, num, minPos, loc - 1);
    }
```

Why loc+1?

Why loc-1?

Would this code work as well on ArrayList?

# Binary Search Analysis

- Each recursive call divides the array in half
- If the array is of size n, it divides (and searches) at most $\log_2 n$ times before the current half is of size $1$
- $O(\log_2 n)$

# Recursion and Backtracking

- All problems considered so far progress steadily towards an answer.
- Consider a maze.  Sometimes you need to “backtrack”.
  - RECURSION makes backtracking easy!
- Idea:
  - 1. Somehow  make a copy of where you are,
  - 2. Try to go forward one step.
    - A. If success,
      - Mark your step on the copy.
      - return to step 1
    - B. If failure
      - throw out copy
      - go some other direction using your original

- Twiddle
  - especially with mazes mark places you have been so you do not retry failed paths

# Do Maze!!

- See Maze.java