Introduction to Programming

Week 7

Magnus Madsen

Week 7: Recursion

Monday

- Recursion
- Pitfalls

Thursday

- Dynamic programming
- Live programming

Prologue

Quote of the Week

"The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination."

— Fred Brooks

Epigram of the Week

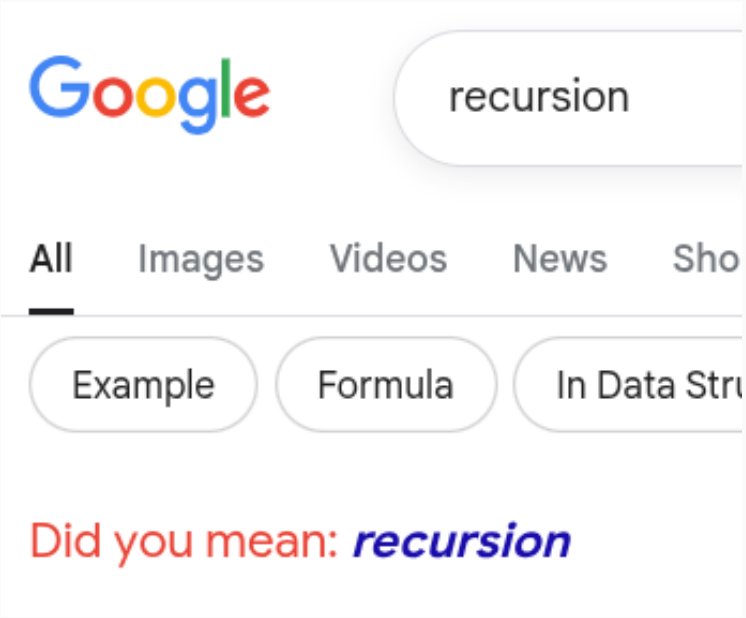
"Prolonged contact with the computer turns mathematicians into clerks and vice versa."

— Alan Perlis

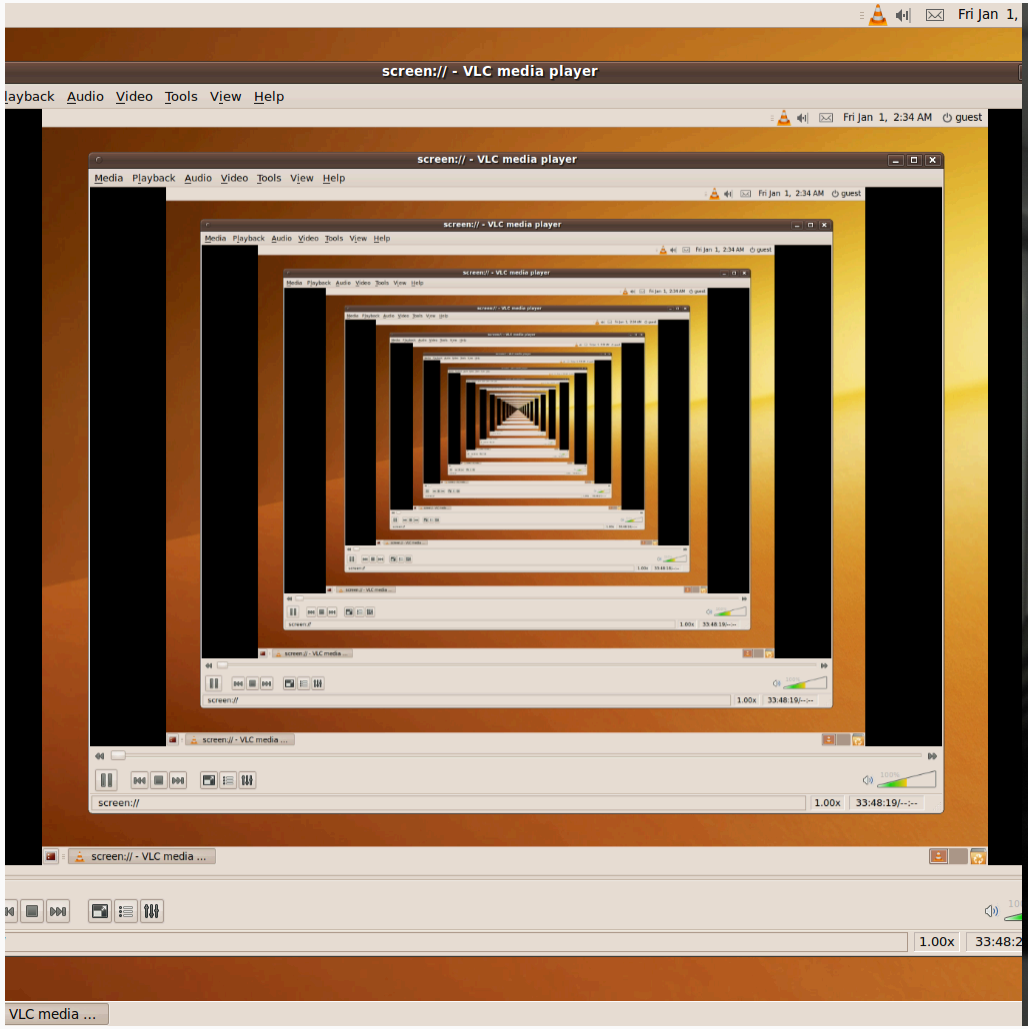
Recursion

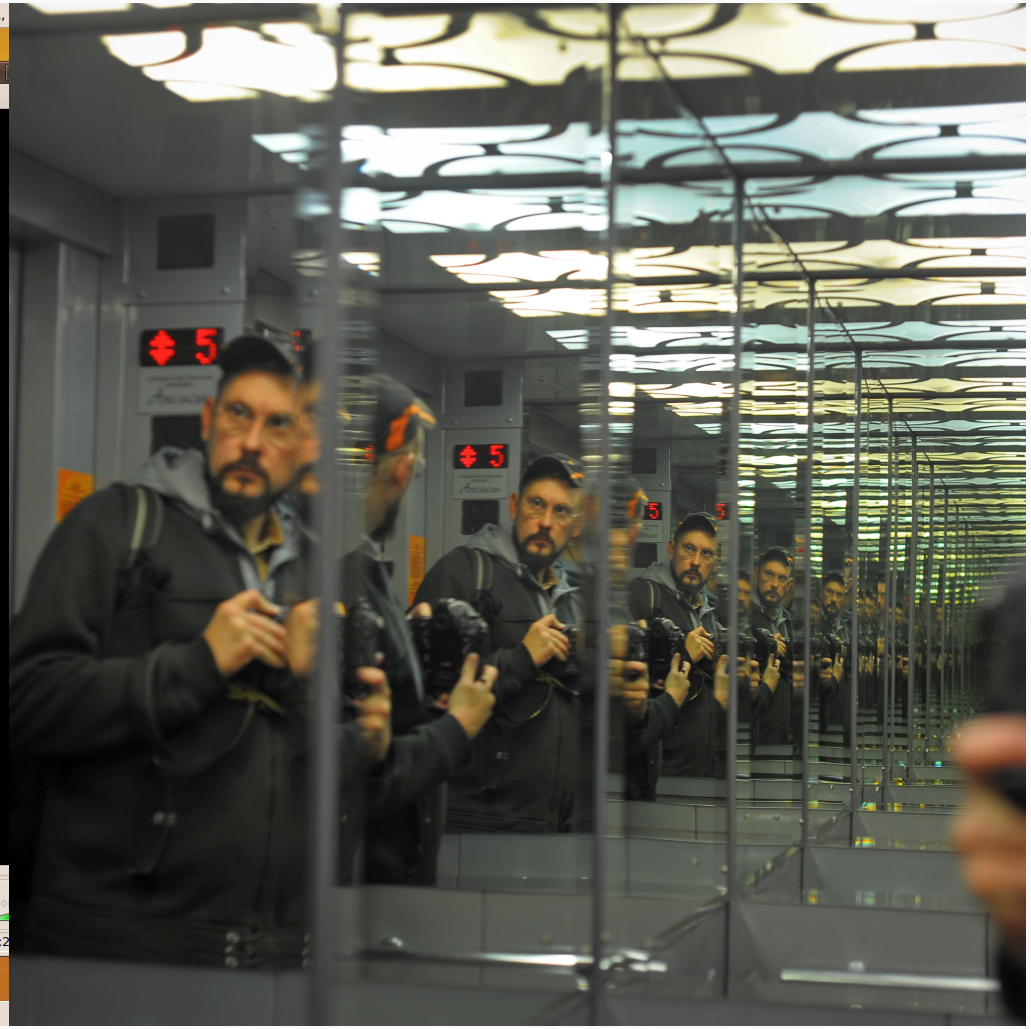
Introduction to Recursion (1/5)

Definition: ***Recursion***. *See Recursion.*

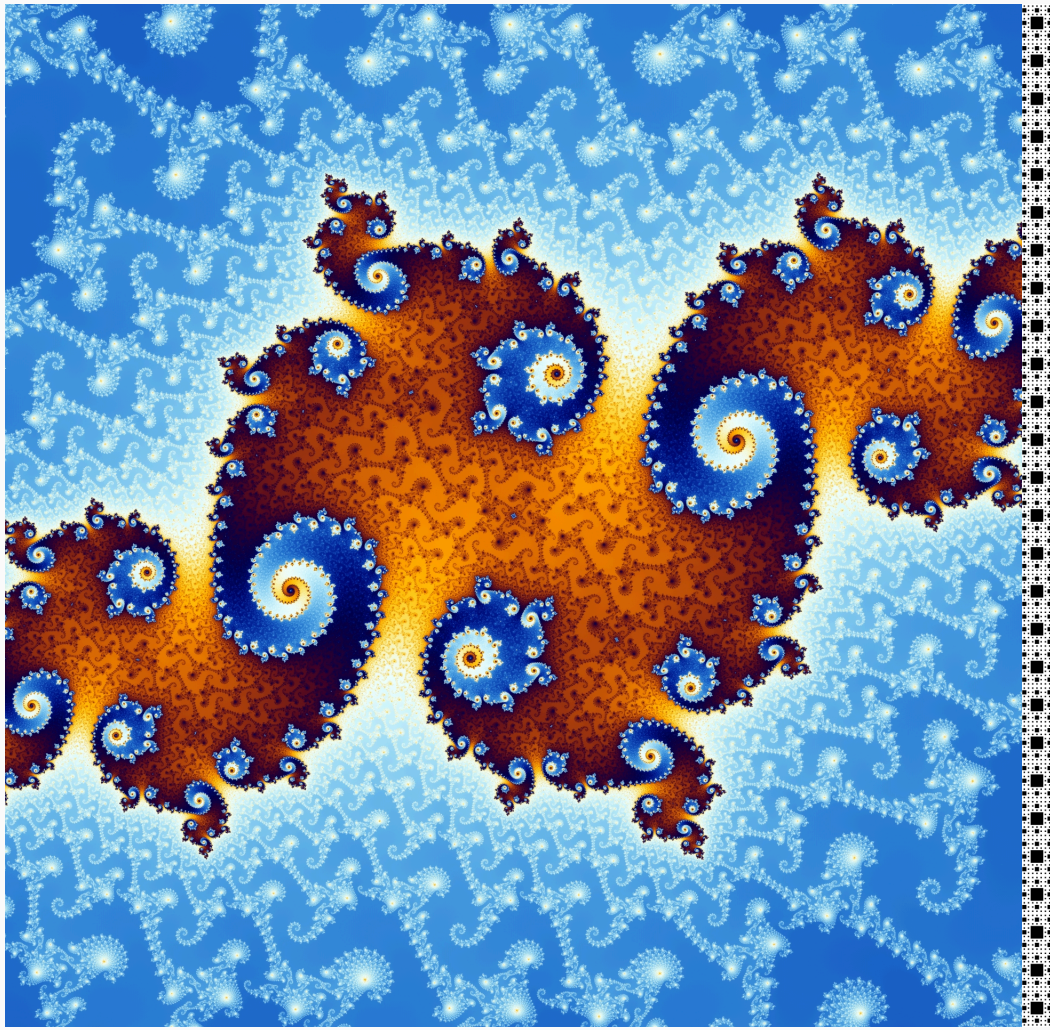


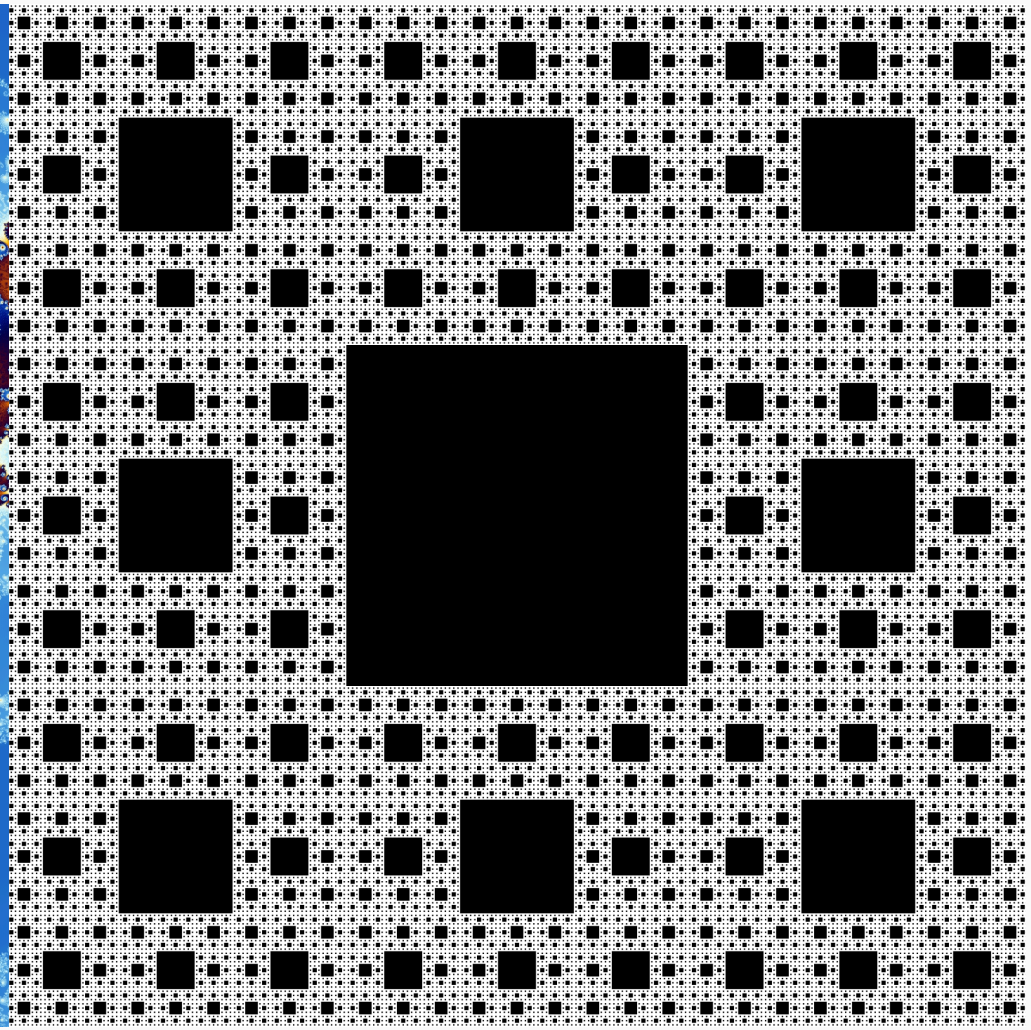
Introduction to Recursion (2/5)





Introduction to Recursion (3/5)





Introduction to Recursion (4/5)

Recursion is everywhere:

- **Biology** - DNA replication, cell division, family trees)
- **Computer Science** (file systems, data structures, algorithms)
- **Mathematics** (Fibonacci sequence, factorials, induction)
- **Language** (nested sentences, grammar rules)
- **Nature** (fractals, snowflakes, coastlines, ferns)

Recursion is the **foundation** for **mathematical induction**.

- Used to prove properties of inductively defined structures.

Recursion: When a function calls *itself*.

You might be thinking: *What in the blazes? How can this be meaningful?*

Why Recursion?

"Recursive programs are often more compact and easier to understand than their nonrecursive counterparts"

— Introduction to Programming in Java

Why Recursion?

> “Recursive programs are often more compact and easier to understand than their nonrecursive counterparts”
>
> — Introduction to Programming in Java

This is true, but to benefit from a powerful tool, you need to know how to use it.

Why Recursion?

"Recursive programs are often more compact and easier to understand than their nonrecursive counterparts"

— Introduction to Programming in Java

This is true, but to benefit from a powerful tool, you need to know how to use it.

"I don't need to learn how to drive because I already know how to walk!"

Example: The Factorial function

The **factorial function** computes the product of all positive integers from 1 to n.

For example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Example: The Factorial function

The **factorial function** computes the product of all positive integers from 1 to n.

For example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

We can implement factorial using a recursive Java method:

```
static long factorial(int n) {
    if (n <= 1) return 1;           // base case
    return n * factorial(n - 1);    // recursive case
}
```

Tracing `factorial`

We can trace the execution of `factorial(5)`:

```
factorial(5)
├─ 5 * factorial(4)
   ├─ 4 * factorial(3)
      ├─ 3 * factorial(2)
         ├─ 2 * factorial(1)
            └─ 1  (base case)
         └─ 2 * 1 = 2
      └─ 3 * 2 = 6
   └─ 4 * 6 = 24
└─ 5 * 24 = 120
```

The function makes **5 recursive calls** before reaching the base case, then **unwinds** the call stack, multiplying the results as it returns.

Recursive Algorithm

Every recursive algorithm has two essential parts:

- **A base case**: A condition that stops the recursion and provides a direct answer.
- **A recursive step**: A recursive call where the argument is a "smaller" or "simpler" version of the original problem, ensuring progress toward the base case.

Recursive Algorithm

Every recursive algorithm has two essential parts:

- **A base case**: A condition that stops the recursion and provides a direct answer.
- **A recursive step**: A recursive call where the argument is a "smaller" or "simpler" version of the original problem, ensuring progress toward the base case.

In `factorial(int n)`:

- the base case is: `if (n <= 1) return 1`
- the recursive step is: `return n * factorial(n - 1)`
  - (calls itself with `n - 1`, a smaller value)

Note: A function can have multiple base cases and/or recursive steps.

Example: Sum of Digits

The **sum of digits** function computes the sum of all digits in a positive integer.

For example: `sumDigits(123) = 1 + 2 + 3 = 6`

Example: Sum of Digits

The **sum of digits** function computes the sum of all digits in a positive integer.

For example: `sumDigits(123) = 1 + 2 + 3 = 6`

We can implement this recursively by extracting the last digit and recursing on the remaining digits:

```
static int sumDigits(int n) {
    if (n == 0) return 0;                  // base case
    return n % 10 + sumDigits(n / 10);     // recursive case
}
```

Note: The recursive call is always on a **smaller** value: `n / 10` removes the last digit, guaranteeing the algorithm will eventually reach the base case where `n = 0`.

Tracing `sumDigits`

We can trace the execution of `sumDigits(123)`:

```
sumDigits(123)
├─ 3 + sumDigits(12)       // 123 % 10 = 3, 123 / 10 = 12
   ├─ 2 + sumDigits(1)     //  12 % 10 = 2,  12 / 10 = 1
      ├─ 1 + sumDigits(0)  //   1 % 10 = 1,   1 / 10 = 0
         └─ 0              (base case: n == 0)
      └─ 1 + 0 = 1
   └─ 2 + 1 = 3
└─ 3 + 3 = 6
```

The function makes **4 recursive calls** before reaching the base case, then unwinds the call stack, summing the digits as it returns: **6**.

Example: Array Sum

The **array sum** function computes the sum of all elements in an array recursively.

For example: `arraySum([3, 5, 2], 0)` returns `3 + 5 + 2 = 10`

Example: Array Sum

The **array sum** function computes the sum of all elements in an array recursively.

For example: `arraySum([3, 5, 2], 0)` returns `3 + 5 + 2 = 10`

We traverse the array by incrementing the index in each recursive call:

```
static int arraySum(int[] arr, int index) {
    if (index == arr.length) return 0;              // base case
    return arr[index] + arraySum(arr, index + 1);   // recursive case
}
```

Note: The recursive call advances through the array with `index + 1`, guaranteeing we'll eventually reach the base case when `index == arr.length`.

Tracing `arraySum`

We can trace the execution of `arraySum([3, 5, 2], 0)`:

```
arraySum([3, 5, 2], 0)
├─ 3 + arraySum([3, 5, 2], 1)
   ├─ 5 + arraySum([3, 5, 2], 2)
      ├─ 2 + arraySum([3, 5, 2], 3)
         └─ 0                              (base case: index == arr.length)
      └─ 2 + 0 = 2
   └─ 5 + 2 = 7
└─ 3 + 7 = 10
```

The function makes **4 recursive calls** to traverse the entire array, then unwinds the call stack, summing elements as it returns: **10**.

Example: Find Maximum

The **find maximum** function finds the largest element in an array recursively.

For example: `findMax([3, 8, 2, 5], 0)` returns `8`

Example: Find Maximum

The **find maximum** function finds the largest element in an array recursively.

For example: `findMax([3, 8, 2, 5], 0)` returns `8`

We compare the current element with the maximum of the remaining array:

```
static int findMax(int[] arr, int index) {
    if (index == arr.length - 1) return arr[index];      // base case
    int maxOfRest = findMax(arr, index + 1);             // recursive case
    return Math.max(arr[index], maxOfRest);
}
```

Note: The base case returns the last element when `index == arr.length - 1`. Each recursive call compares the current element with the maximum of all elements after it.

Tracing `findMax`

We can trace the execution of `findMax([3, 8, 2, 5], 0)`:

```
findMax([3, 8, 2, 5], 0)
├─ findMax([3, 8, 2, 5], 1)
   ├─ findMax([3, 8, 2, 5], 2)
      ├─ findMax([3, 8, 2, 5], 3)
         └─ 5                          (base case: last element)
      └─ max(2, 5) = 5
   └─ max(8, 5) = 8
└─ max(3, 8) = 8
```

The function makes **4 recursive calls** to reach the base case, then unwinds comparing elements pairwise, ultimately returning the maximum: **8**.

Q:

Fill in the blanks:

```
static int arrayProduct(int[] arr, int index) {
    if (index == 0) return _______________;  // base case
    return arr[index - 1] * _______________;  // recursive call
}

// Initial call: arrayProduct(arr, arr.length)
```

Example: Binary Search

The **binary search** algorithm finds a target value in a **sorted** array by repeatedly dividing the search space in half.

For example: searching for `7` in `[1, 3, 5, 7, 9, 11]` returns index `3`

Example: Binary Search

The **binary search** algorithm finds a target value in a **sorted** array by repeatedly dividing the search space in half.

For example: searching for `7` in `[1, 3, 5, 7, 9, 11]` returns index `3`

We compare the target with the middle element and recurse on the appropriate half:

```
static int binarySearch(int[] arr, int target, int low, int high) {
    if (low > high) return -1;                          // base case: not found
    int mid = (low + high) / 2;
    if (arr[mid] == target) return mid;                 // base case: found
    if (target < arr[mid])
        return binarySearch(arr, target, low, mid - 1);    // search left half
    else
        return binarySearch(arr, target, mid + 1, high);   // search right half
}
```

Note: Each recursive call eliminates half the search space, making this algorithm very efficient with **O(log n)** time complexity.

Tracing `binarySearch`

We can trace the execution of `binarySearch([1, 3, 5, 7, 9, 11], 7, 0, 5)`:

```
binarySearch([1, 3, 5, 7, 9, 11], 7, 0, 5)
├─ mid = 2, arr[2] = 5, 7 > 5
└─ binarySearch([1, 3, 5, 7, 9, 11], 7, 3, 5)
   ├─ mid = 4, arr[4] = 9, 7 < 9
   └─ binarySearch([1, 3, 5, 7, 9, 11], 7, 3, 3)
      ├─ mid = 3, arr[3] = 7, 7 == 7
      └─ 3                          (base case: found at index 3)
```

The algorithm makes **3 recursive calls**, each time halving the search space, before finding the target at index **3**.

Example: Euclid's Algorithm

The **greatest common divisor** of two numbers a and b is the largest positive integer r that divides both numbers, i.e. $a \bmod r = 0$ and $b \bmod r = 0$.

For example: $\gcd(48, 18) = 6$ because 6 is the largest number that divides both 48 and 18.

Example: Euclid's Algorithm

The **greatest common divisor** of two numbers a and b is the largest positive integer r that divides both numbers, i.e. $a \bmod r = 0$ and $b \bmod r = 0$.

For example: $\gcd(48, 18) = 6$ because 6 is the largest number that divides both 48 and 18.

We can use **Euclid's** algorithm to compute the greatest common divisor:

```
static int gcd(int a, int b) {      // where a >= b
    if (b == 0) return a;           // base case
    return gcd(b, a % b);           // recursive case
}
```

Note: The recursive call is always on a **smaller** value: `a % b` is always less than `b`, guaranteeing the algorithm will eventually reach the base case where `b = 0`.

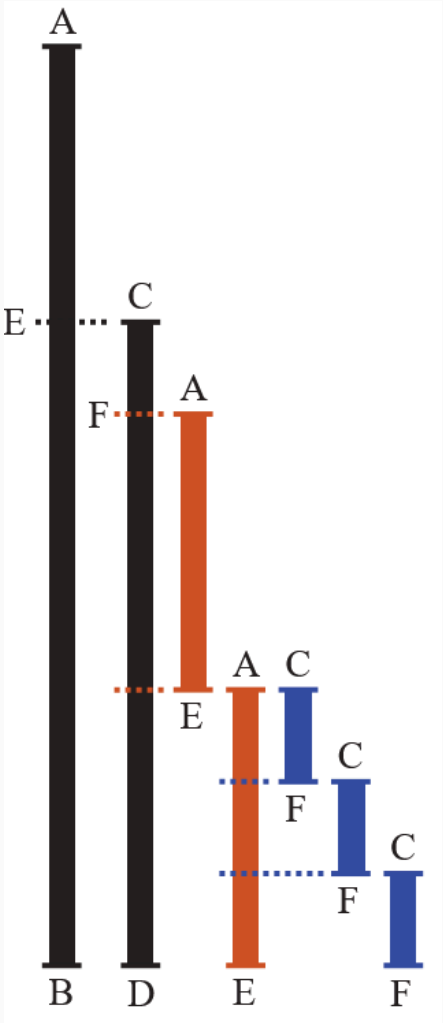
Tracing `gcd`

We can trace the execution of `gcd(48, 18)`:

```
gcd(48, 18)
├─ gcd(18, 12)      // 48 % 18 = 12
   ├─ gcd(12, 6)    // 18 % 12 = 6
      ├─ gcd(6, 0)  // 12 % 6 = 0
         └─ 6       (base case: b == 0)
      └─ 6
   └─ 6
└─ 6
```

The algorithm makes **4 recursive calls** before reaching the base case when `b = 0`, then returns the greatest common divisor: **6**.

Understanding `gcd`



Pitfalls

Pitfall: Missing Base Case

This function attempts to compute the sum of integers from 1 to n, but **lacks a base case**.

For example, `sum(5)` should return `1 + 2 + 3 + 4 + 5 = 15`.

```
int sum(int n) {
    return n + sum(n - 1);    // infinite recursion!
}
```

Without a base case, this function will call itself **infinitely**, leading to a `StackOverflowError`.

The missing base case should be:

```
if (n <= 0) return 0;
```

Pitfall: No Convergence

This function attempts to count down from n, but the recursive call doesn't make progress toward the base case:

```
int countdown(int n) {
    if (n == 0) return 0;
    return countdown(n);    // calls itself with same value!
}
```

The function should decrease `n` by 1 in each recursive call: `return countdown(n - 1)`

Pitfall: Excessive Memory Requirements (1/2)

This function correctly computes the sum of integers from 1 to n, but will cause a `StackOverflowError` for large values of n:

```
static long sum(int n) {
    if (n <= 0) return 0;
    return n + sum(n - 1);
}
```

Each recursive call adds a new frame to the call stack. For `sum(100000)`, this creates 100,000 stack frames, exceeding memory limits.

Pitfall: Excessive Memory Requirements (2/2)

```
public static void main(String[] args) {
    long x = sum(100000);
}
```

```
Exception in thread "main" java.lang.StackOverflowError
  at Test.sum(Test.java:6)
  at Test.sum(Test.java:6)
  at Test.sum(Test.java:6)
  at Test.sum(Test.java:6)
  at Test.sum(Test.java:6)
  at Test.sum(Test.java:6)...
```

Pitfall: Excessive Recomputation

The Fibonacci sequence computes each number as the sum of the two preceding numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
static int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

This implementation recomputes the same values repeatedly - for example, `fibonacci(5)` calculates `fibonacci(3)` twice and `fibonacci(2)` three times, making the algorithm exponentially slow.

Q:

What can you say about this recursive function?

```
static int countByThrees(int n) {
    if (n == 0) return 0;
    System.out.println(n);
    return 1 + countByThrees(n - 3);
}
```

Collatz Conjecture

Collatz Sequence

Collatz function of *N*.

- If *N* is 1, stop.
- If *N* is even, divide by 2.
- If *N* is odd, multiply by 3 and add 1.

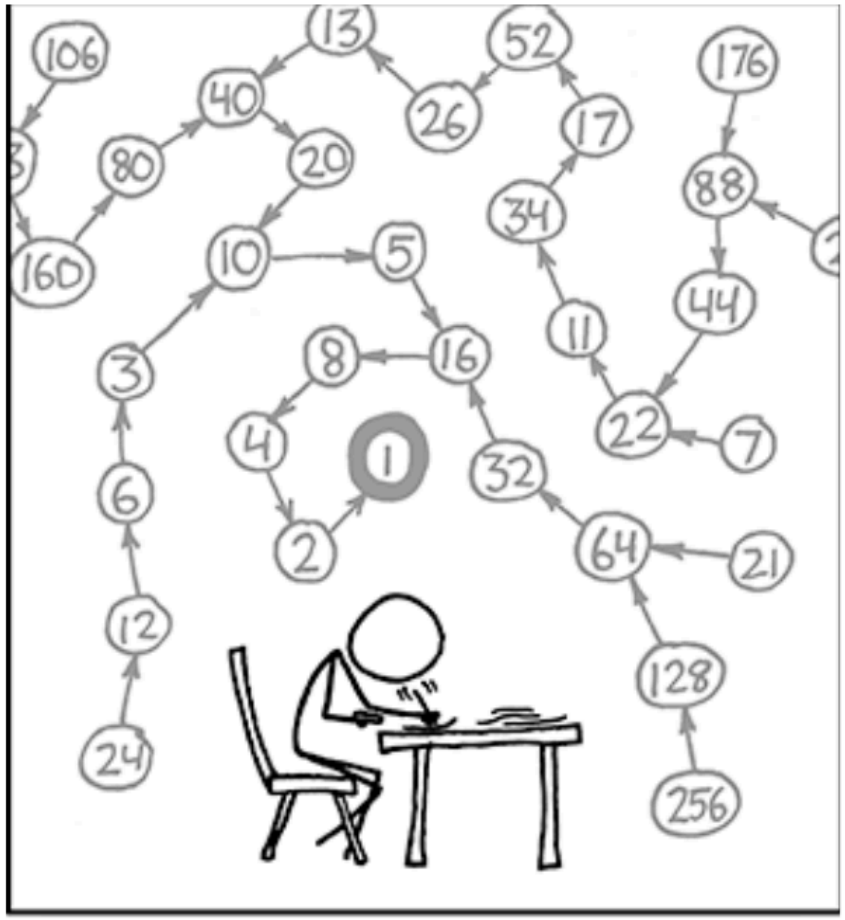
7	22	11	34	17	52	26	13	49	20	...

```
public static void collatz(int N)
{
    StdOut.print(N + " ");
    if (N == 1) return;
    if (N % 2 == 0) collatz(N / 2);
    collatz(3*N + 1);
}
```

```
% java Collatz 7
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Amazing fact. No one knows whether or not this function terminates for all *N* (!)

Note. We usually ensure termination by only making recursive calls for smaller *N*.



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Q:

What does this mystery function compute?

```
static int mystery(int a, int b) { // a, b are non-negative
    if (a == 0) return b;          // base case 1
    if (b == 0) return a;          // base case 2
    if (a > b)
        return mystery(a - b, b);   // recursive call 1
    else
        return mystery(a, b - a);   // recursive call 2
}

// mystery(12, 8) returns ?
// mystery(15, 10) returns ?
// mystery(21, 14) returns ?
```

Drawing a Beautiful Tree

A Beautiful Tree: `main`

```
public static void main(String[] args) {
    // Set up the drawing canvas
    StdDraw.setCanvasSize(800, 800);
    StdDraw.clear(new java.awt.Color(135, 206, 235)); // sky blue background

    // Draw ground
    StdDraw.setPenColor(new java.awt.Color(139, 90, 43)); // brown
    StdDraw.filledRectangle(0.5, 0.05, 0.5, 0.05);

    // Draw grass
    StdDraw.setPenColor(new java.awt.Color(0, 100, 0)); // dark green
    StdDraw.filledRectangle(0.5, 0.08, 0.5, 0.02);

    // Draw the tree starting from bottom center
    double startX = 0.5;
    double startY = 0.1;
    double initialLength = 0.25;
    double initialAngle = 90;  // pointing upward
    int recursionDepth = 10;
    drawBranch(startX, startY, initialLength, initialAngle, recursionDepth);
}
```

A Beautiful Tree: `drawBranch` (1/2)

```java
public static void drawBranch(double x0, double y0, double length, double angle, int depth) {
    if (depth == 0) return;  // base case

    // Calculate end point of current branch
    double x1 = x0 + length * Math.cos(Math.toRadians(angle));
    double y1 = y0 + length * Math.sin(Math.toRadians(angle));

    // Set pen thickness based on depth (thicker trunk, thinner branches)
    StdDraw.setPenRadius(0.002 * depth);

    // Set color (brown for trunk, gradually greener for branches)
    if (depth > 5) {
        StdDraw.setPenColor(new java.awt.Color(101, 67, 33)); // brown
    } else if (depth > 2) {
        StdDraw.setPenColor(new java.awt.Color(34, 139, 34)); // forest green
    } else {
        StdDraw.setPenColor(new java.awt.Color(0, 200, 0)); // bright green
    }

    // Draw the branch
    StdDraw.line(x0, y0, x1, y1);

    // ... (continued next slide) ...
}
```

A Beautiful Tree: `drawBranch` (2/2)

```
public static void drawBranch(double x0, double y0, double length, double angle, int depth) {
    // ... previous slide ...

    // Recursive calls for sub-branches
    double branchingAngle = 25;  // angle between branches
    double shrinkFactor = 0.7;   // how much smaller each branch is

    // Left branch
    drawBranch(x1, y1, length * shrinkFactor, angle + branchingAngle, depth - 1);

    // Right branch
    drawBranch(x1, y1, length * shrinkFactor, angle - branchingAngle, depth - 1);

    // Optional middle branch (for fuller tree)
    if (Math.random() > 0.5) {
        drawBranch(x1, y1, length * shrinkFactor * 0.8, angle + (Math.random() - 0.5) * 10, depth - 1);
    }
}
```

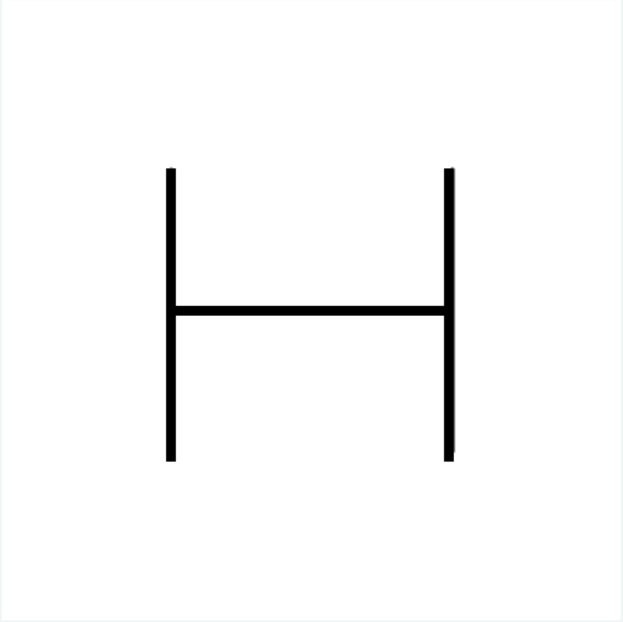
H-Trees

"Hello, World" of recursive graphics: H-trees

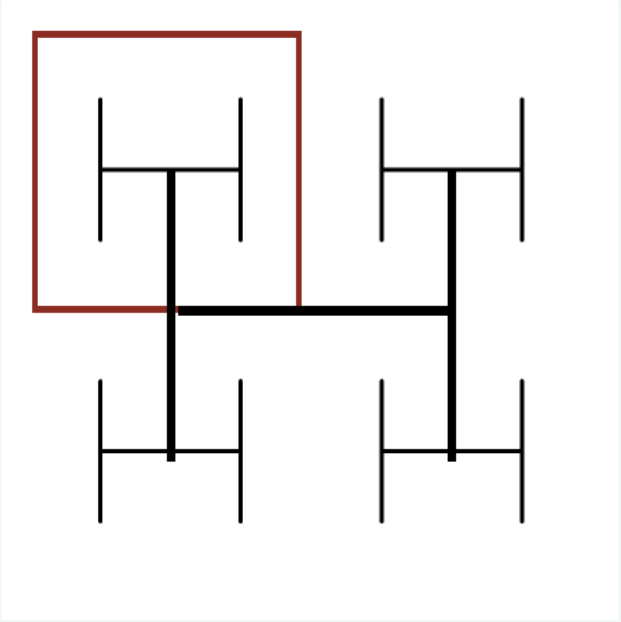
H-tree of order *n*

- If *n* is 0, do nothing.
- Draw an H, centered.
- Draw four H-trees of order *n* −1 and half the size, centered at the tips of the H.

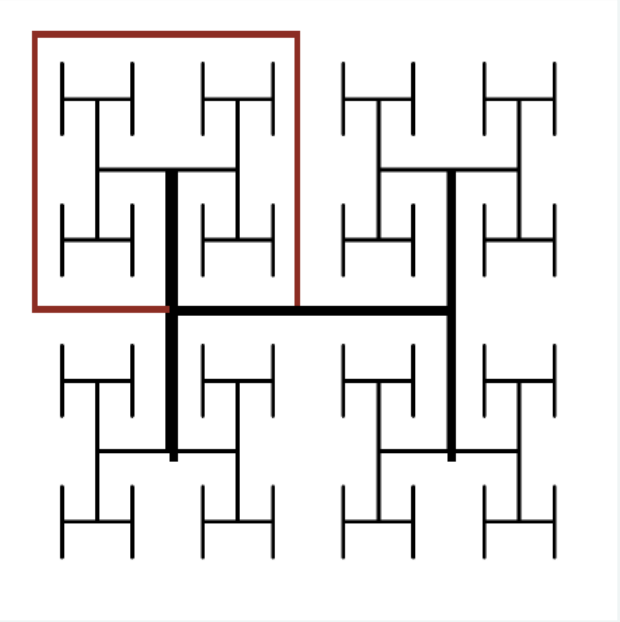
order 1



order 2

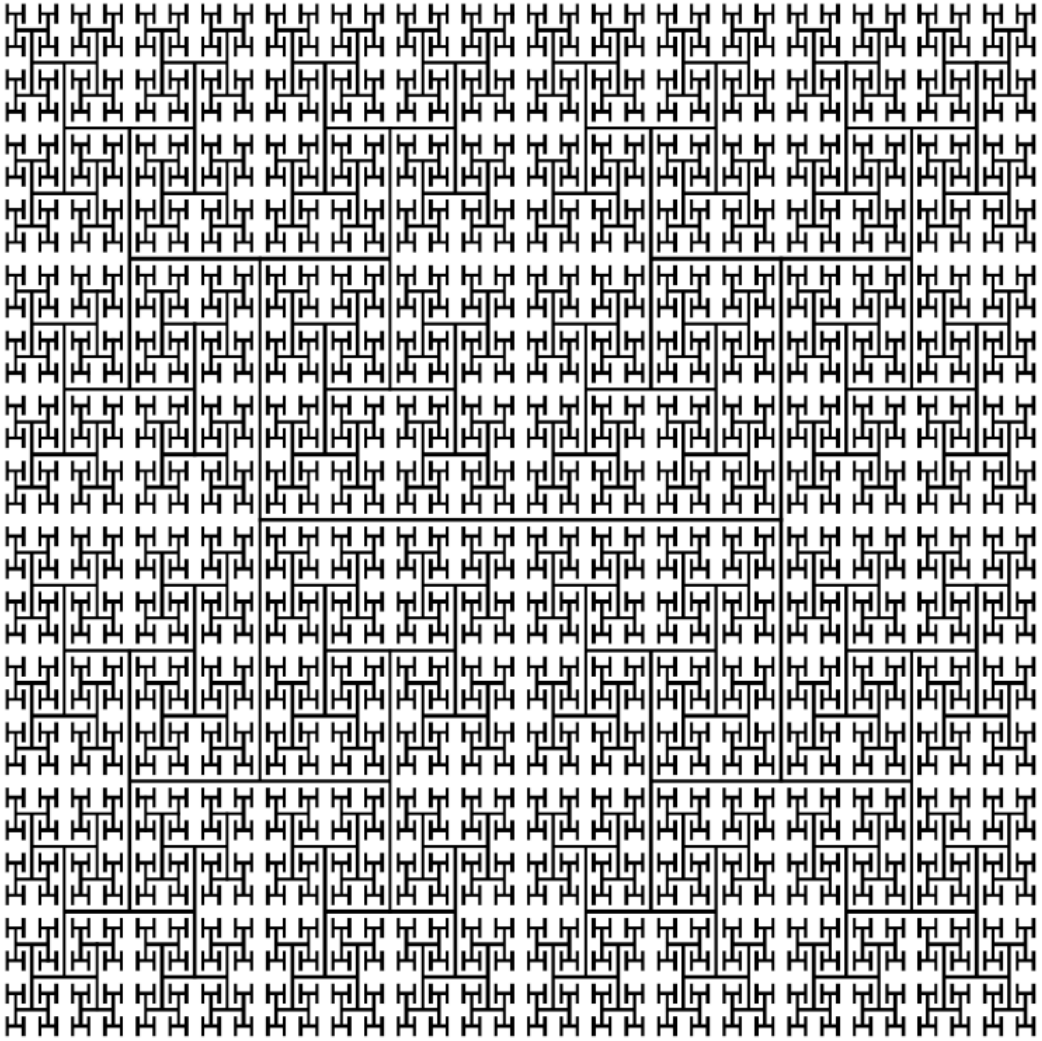


order 3



H-trees

Application. Connect a large set of regularly spaced sites to a single source.



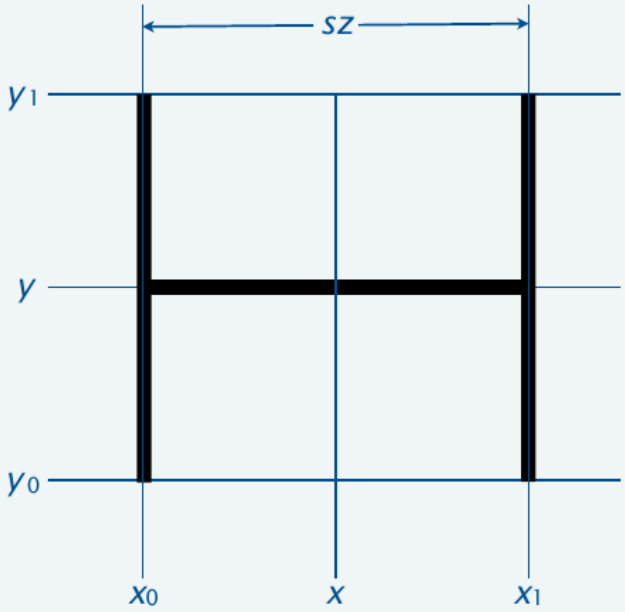
order 6

Recursive H-tree implementation

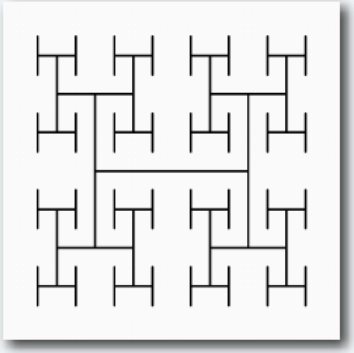
```
public class Htree
{
   public static void draw(int n, double sz, double x, double y)
   {
      if (n == 0) return;
      double x0 = x - sz/2, x1 = x + sz/2;
      double y0 = y - sz/2, y1 = y + sz/2;
      StdDraw.line(x0,  y, x1,  y);
      StdDraw.line(x0, y0, x0, y1);
      StdDraw.line(x1, y0, x1, y1);
      draw(n-1, sz/2, x0, y0);
      draw(n-1, sz/2, x0, y1);
      draw(n-1, sz/2, x1, y0);
      draw(n-1, sz/2, x1, y1);
   }
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      draw(n, .5, .5, .5);
   }
}
```

draw the H, centered on (*x*, *y*)

draw four half-size H-trees




```
% java Htree 3
```



Towers of Hanoi

Towers of Hanoi puzzle

A legend of uncertain origin

- $n = 64$ discs of differing size; 3 posts; discs on one of the posts from largest to smallest.
- An ancient prophecy has commanded monks to move the discs to another post.
- When the task is completed, *the world will end.*

Rules

- Move discs one at a time.
- Never put a larger disc on a smaller disc.

Q. Generate list of instruction for monks ?

Q. When might the world end ?

n = 10

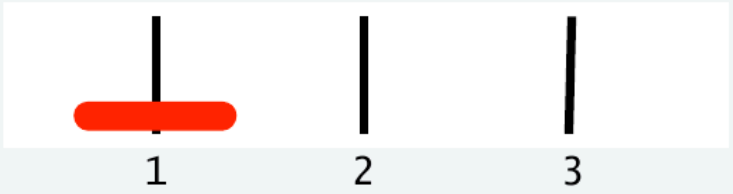
before

after

Towers of Hanoi

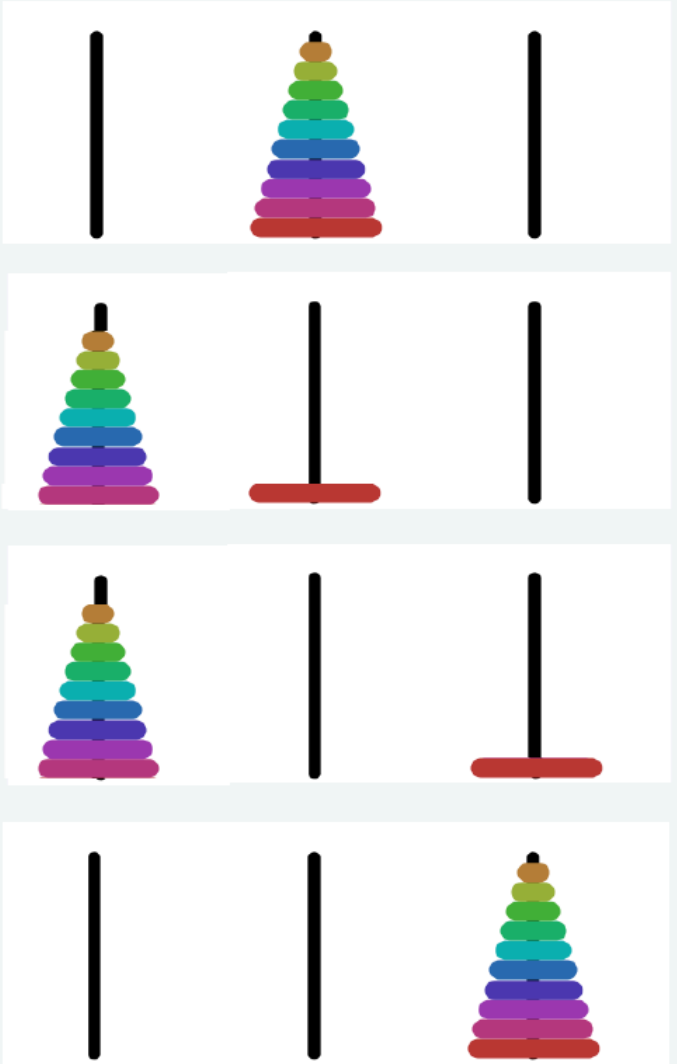
For simple instructions, use cyclic wraparound

- Move *right* means 1 to 2, 2 to 3, or 3 to 1.
- Move *left* means 1 to 3, 3 to 2, or 2 to 1.

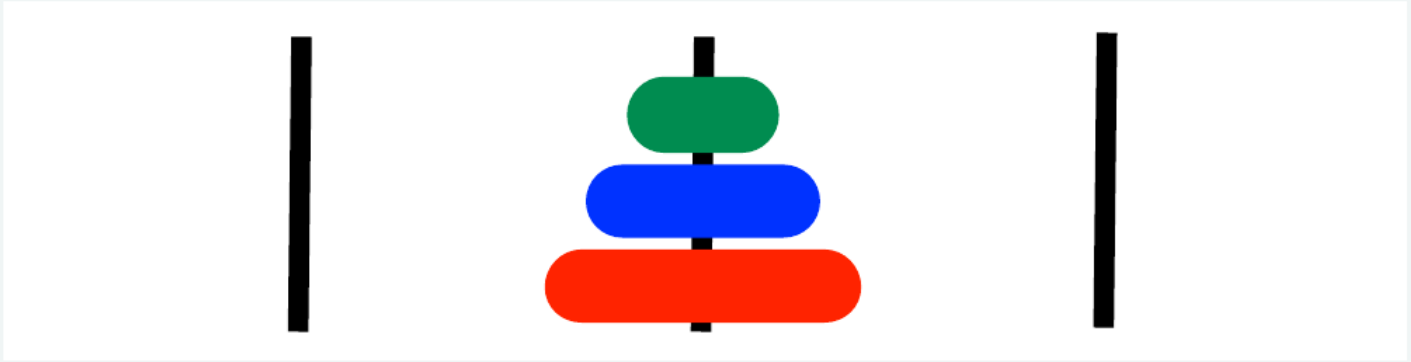


A recursive solution

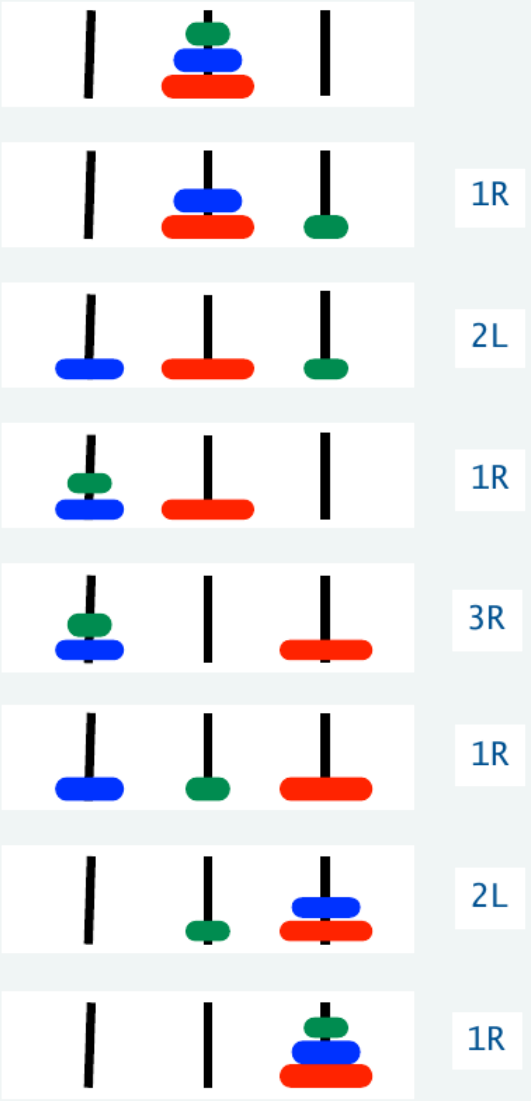
- Move $n - 1$ discs to the left (recursively).
- Move largest disc to the *right*.
- Move $n - 1$ discs to the left (recursively).



Towers of Hanoi solution (n = 3)



1R	2L	1R	3R	1R	2L	1R



Towers of Hanoi: recursive solution

`hanoi(n)`: Print moves for n discs.

- Return one space for $n = 0$.
- Otherwise, set `move` to the specified move for disc n.
- Then sandwich `move` between two copies of `hanoi(n-1)`.

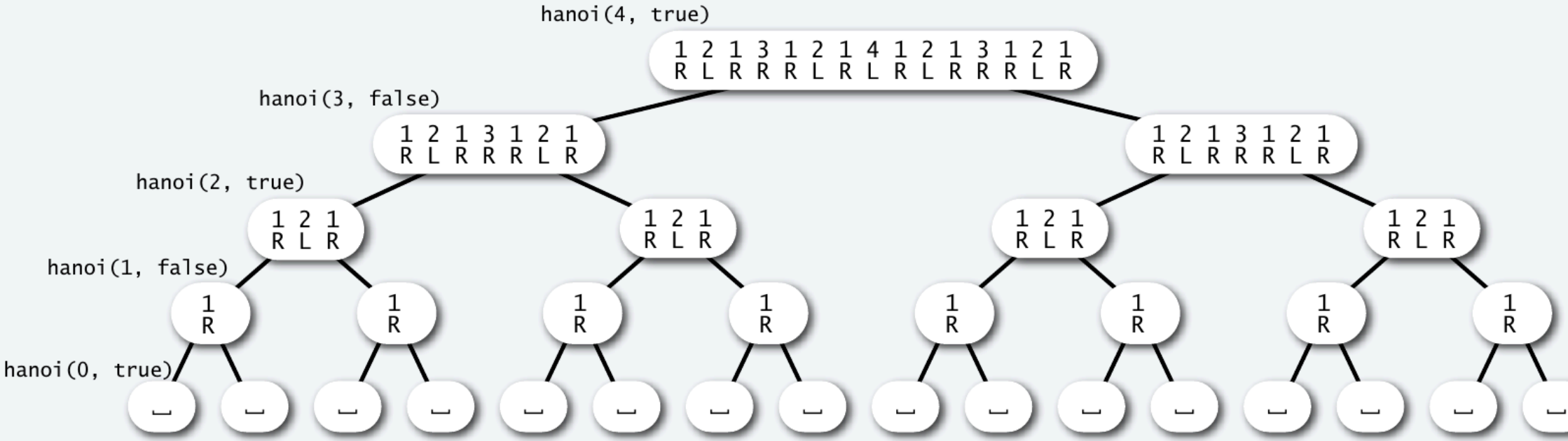
```
public class Hanoi
{
   public static String hanoi(int n, boolean left)
   {
      if (n == 0) return " ";
      String move;
      if (left) move = n + "L";
      else      move = n + "R";
      return hanoi(n-1, !left) + move + hanoi(n-1, !left);
   }

   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      StdOut.println(hanoi(n, false));
   }
}
```

```
% java Hanoi 3
 1R 2L 1R 3R 1R 2L 1R
```

Recursive call tree for towers of Hanoi

Structure is the *same* as for the ruler function and suggests 3 useful and easy-to-prove facts.

- Each disc always moves in the same direction.
- Moving smaller disc always alternates with a unique legal move.
- Moving n discs requires $2^n - 1$ moves.



Answers for towers of Hanoi

Q. Generate list of instructions for monks ?

A. (Long form). 1L 2R 1L 3L 1L 2R 1L 4R 1L 2R 1L 3L 1L 2R 1L 5L 1L 2R 1L 3L 1L 2R 1L 4R ...

A. (Short form). Alternate "1L" with the only legal move not involving the disc 1.

"L" or "R" depends on whether *n* is odd or even

Q. When might the world end ?

A. Not soon: need $2^{64} - 1$ moves.

moves per second	*end of world*
1	5.84 billion centuries
1 billion	5.84 centuries

Note: Recursive solution has been proven optimal.

Q: (True or False): The `binarySearch` method can cause a `StackOverflowException`?

Short programs, Long time.

We have seen that the Towers of Hanoi takes an **exponentially** long time to compute.

With recursion, we can easily write programs that appear simple but run for **extremely** long periods.

Upshot: We must **think carefully** about algorithmic complexity.

Later we shall see that dynamic programming can sometimes help speed things up.

Dynamic Programming

Fibonacci numbers

Let $F_n = F_{n-1} + F_{n-1}$ for $n > 1$ with $F_0 = 0$ and $F_1 = 1$.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	...

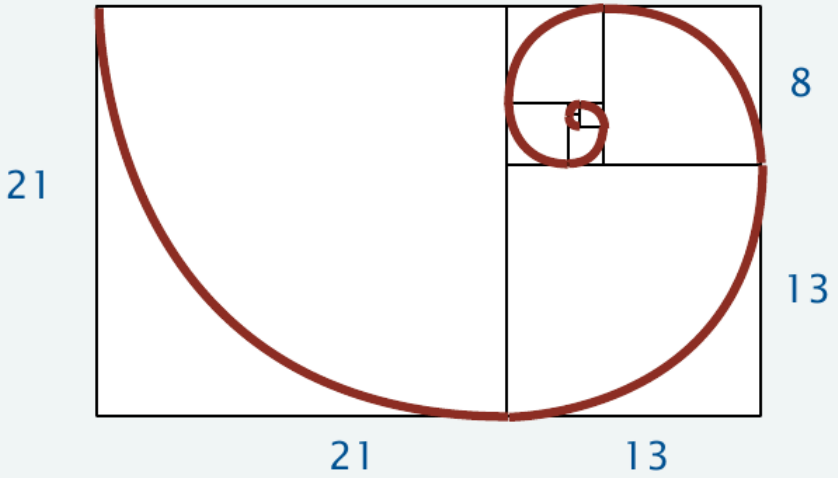
Models many natural phenomena and is widely found in art and architecture.

Leonardo Fibonacci
c. 1170 – c. 1250

Examples.

- Model for reproducing rabbits.
- Nautilus shell.
- Mona Lisa.
- ...

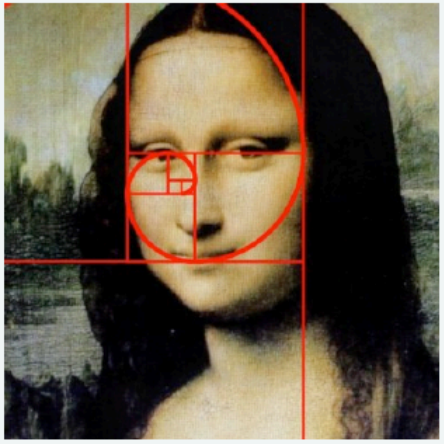
golden ratio F_n / F_{n-1}

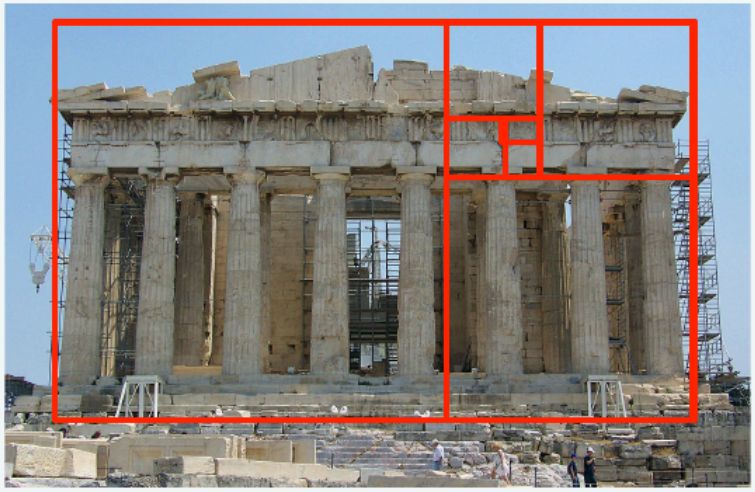


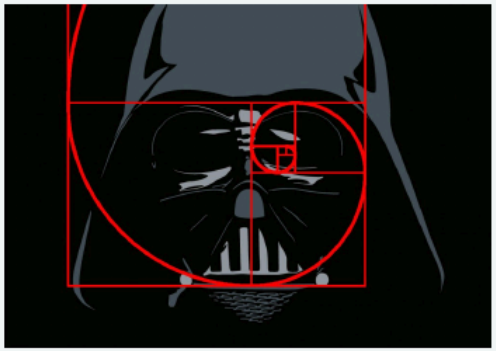
Facts (known for centuries).

- $F_n / F_{n-1} \to \Phi = 1.618...$ as $n \to \infty$
- F_n is the closest integer to $\Phi^n/\sqrt{5}$

Fibonacci numbers and the golden ratio in the wild







1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1

1
1
2
3
5
8
13
21

Computing Fibonacci numbers

Q. [Curious individual.] What is the exact value of F_{60} ?

A. [Novice programmer.] Just a second. I'll write a recursive program to compute it.

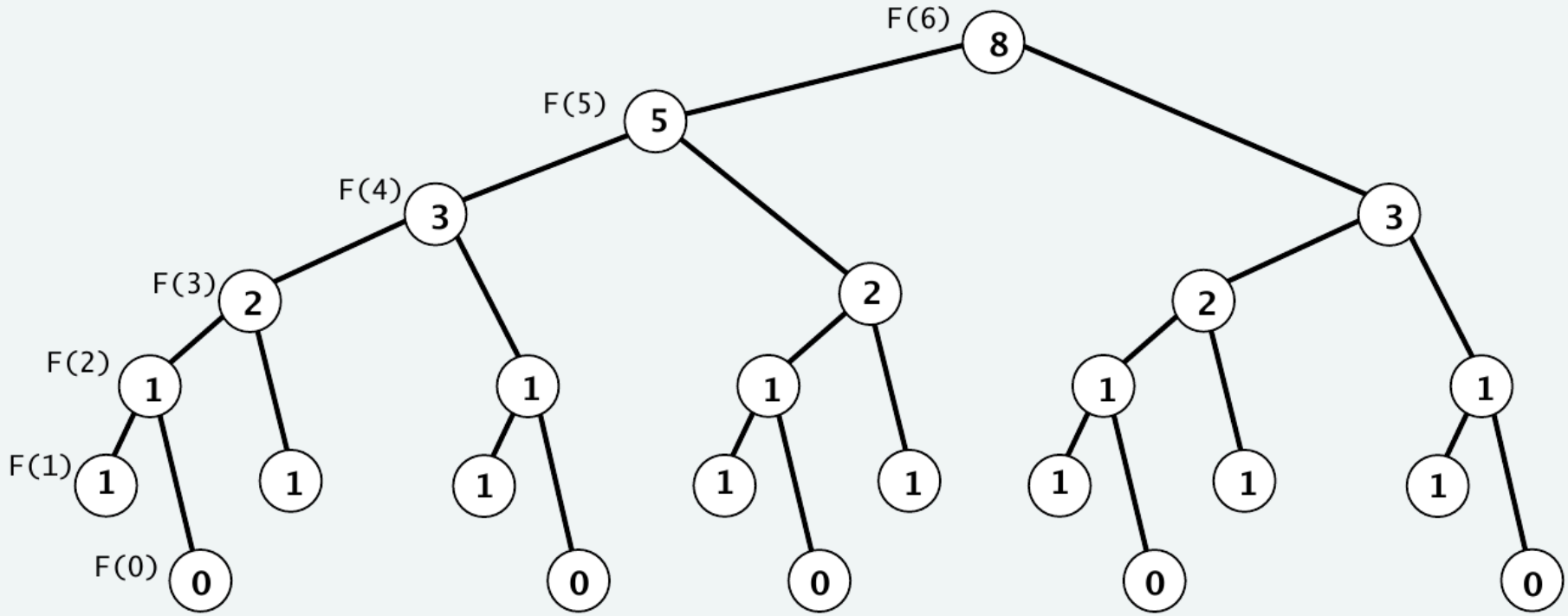
```
public class FibonacciR
{
   public static long F(int n)
   {
      if (n == 0) return 0;
      if (n == 1) return 1;
      return F(n-1) + F(n-2);
   }
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      StdOut.println(F(n));
   }
}
```

```
% java FibonacciR 5
5
% java FibonacciR 6
8
% java FibonacciR 10
55
% java FibonacciR 12
144
% java FibonacciR 50
12586269025
% java FibonacciR 60
```

takes a few minutes
Hmmm. Why is that?

Is something wrong with my computer?

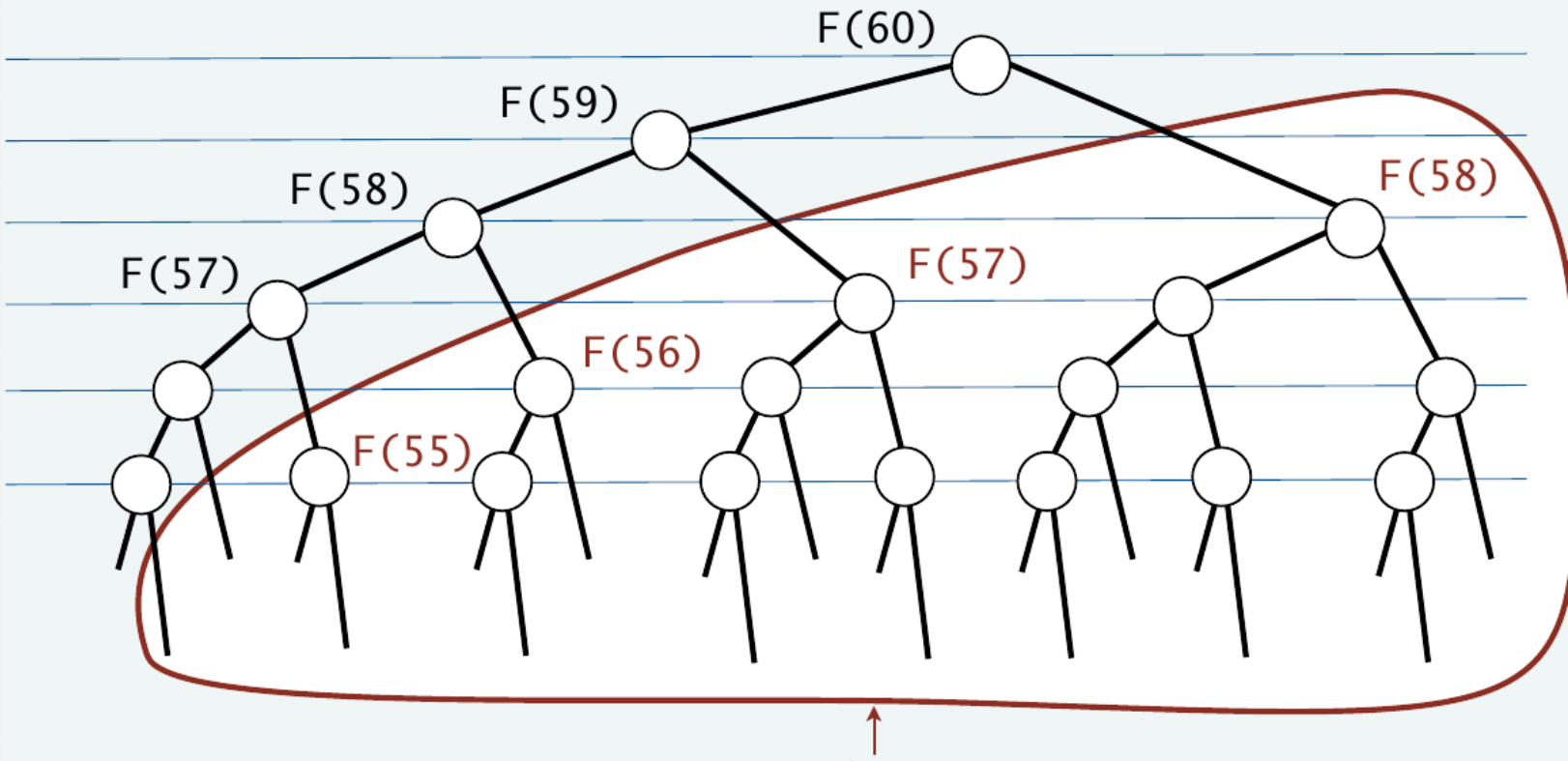
Recursive call tree for Fibonacci numbers



Exponential waste

Let C_n be the number of times F(n) is called when computing F(60).

n	C_n	
60	1	F_1
59	1	F_2
58	2	F_3
57	3	F_4
56	5	F_5
55	8	F_6
	...	...
0	>2.5×10^12	F_{61}

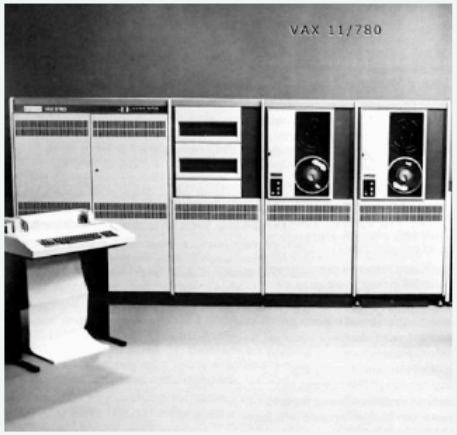


Exponentially wasteful to recompute all these values.
(trillions of calls on F(0), not to mention calls on F(1), F(2),...)

Exponential waste dwarfs progress in technology

If you engage in exponential waste, you *will not* be able to solve a large problem.

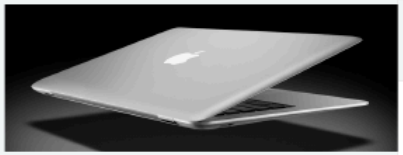
1970s



VAX 11/780

n	*time to compute* F_n
30	minutes
40	hours
50	weeks
60	years
70	centuries
80	millenia

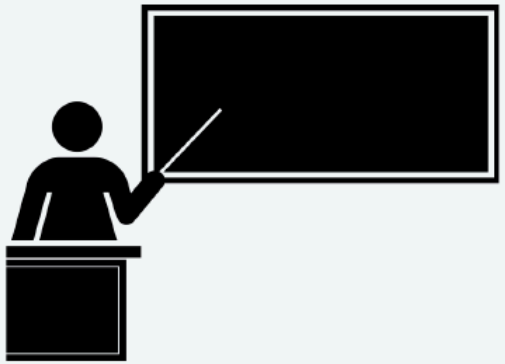
2010s: 10,000+ times faster


Macbook Air

n	*time to compute* F_n
50	minutes
60	hours
70	weeks
80	years
90	centuries
100	millenia

1970s: "That program won't compute F_{60} before you graduate! "

2010s: "That program won't compute F_{80} before you graduate! "



Introduction to Dynamic Programming

Dynamic Programming is an optimization technique for recursive algorithms.

Key idea: **Avoid recomputation** by **storing results of subproblems**.

Two Approaches to Dynamic Programming

Bottom-Up (Tabulation):

- Start with smallest subproblems
- Build up to the original problem
- Use iterative approach with arrays/tables

Top-Down (Memoization):

- Start with original problem
- Cache results as we compute them
- Use recursive approach with memory

We will focus on **Top-Down (Memoization)** because it uses recursion.

Example: Fibonacci with Memoization

Without memoization: `fib(n)` takes **exponential time:**

```
static int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Example: Fibonacci with Memoization (cont.)

With memoization: `fib(n)` takes **linear time:**

```
// a global variable
static int[] memo = new int[100];

static int fibMemo(int n) {
    if (n <= 1) return n;
    if (memo[n] != 0) return memo[n];
    memo[n] = fibMemo(n-1) + fibMemo(n-2);
    return memo[n];
}
```

Example: Coin Change Problem

The **coin change problem** is the following: Given **coin denominations** and a **target amount**, find the minimum number of coins that sum up to that amount.

Example: Make 11¢ using coins [1¢, 5¢, 6¢, 9¢]

- Solution: 2 coins (5¢ + 6¢)
- Without dynamic programming: **exponential time** - recomputes same amounts.
- With dynamic programming: **polynomial time** - each amount computed once.

Coin Change without Memoization

Without memoization: `minCoins(amount)` takes **exponential time:**

```
static int minCoins(int[] coins, int amount) {
    if (amount == 0) return 0;
    if (amount < 0) return -1;

    int min = Integer.MAX_VALUE;
    for (int i = 0; i < coins.length; i++) {
        int result = minCoins(coins, amount - coins[i]);
        if (result >= 0) {
            min = Math.min(min, result + 1);
        }
    }
    if (min == Integer.MAX_VALUE) {
        return -1;
    } else {
        return min;
    }
}
```

Tracing `minCoins`

Trace of `minCoins([1, 5, 6], 7)`:

```
minCoins(7)
  ├─ minCoins(6) [using 1¢]
  │    ├─ minCoins(5) → ... → 1
  │    ├─ minCoins(1) → ... → 1
  │    └─ minCoins(0) → 0          ✓ min = 1
  ├─ minCoins(2) [using 5¢]
  │    ├─ minCoins(1) → ... → 1
  │    └─ ... (negative amounts)   ✓ min = 2
  └─ minCoins(1) [using 6¢]
       └─ minCoins(0) → 0          ✓ min = 1
Returns: 2
```

Note: `minCoins(1)` is computed multiple times!

Coin Change with Memoization

With memoization: `minCoins(amount)` takes **polynomial time:**

```
// a global variable
static int[] memo = new int[100];

static int minCoins(int[] coins, int amount) {
    if (amount == 0) return 0;
    if (amount < 0) return -1;
    if (memo[amount] != 0) return memo[amount];
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < coins.length; i++) {
        int result = minCoins(coins, amount - coins[i]);
        if (result >= 0) {
            min = Math.min(min, result + 1);
        }
    }
    if (min == Integer.MAX_VALUE) {
        memo[amount] = -1;
    } else {
        memo[amount] = min;
    }
    return memo[amount];
}
```

When to Use Dynamic Programming

Use when a problem has:

1. **Overlapping subproblems** - same subproblems computed multiple times
2. **Optimal substructure** - optimal solution contains optimal solutions to subproblems

Examples: Fibonacci, longest common subsequence, shortest paths, ...

More advanced techniques covered in data structures and algorithms course.

Q: (True or False): If a recursive function has two base cases it must have two recursive calls?

Live Programming

Live Programming

- FindMax
  - Inspect the call stack.
  - See a `StackOverflowException`
- Fibonacci
- Fibonacci with Memoization

Quick Midway Evaluation

Course Difficulty (1/5)

On the one hand:

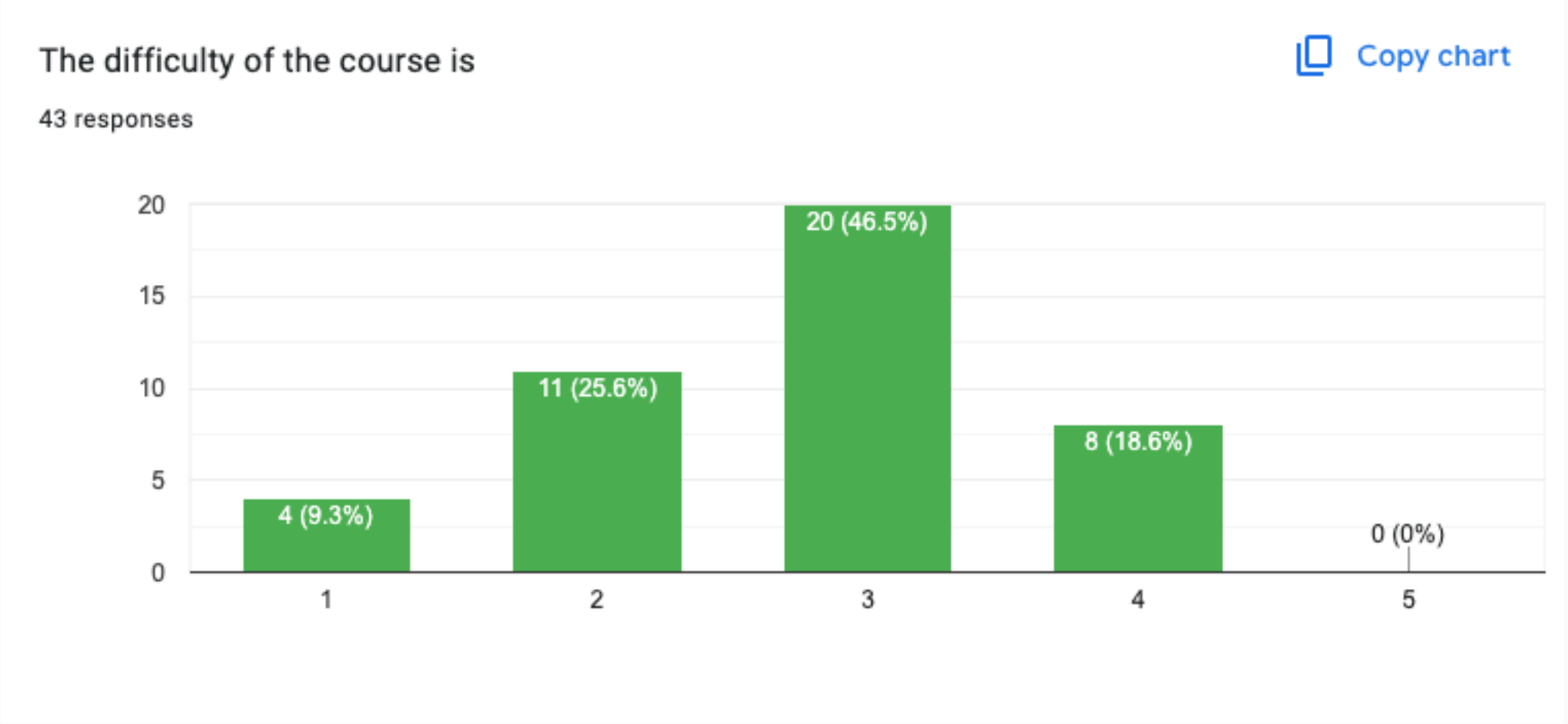
- “It is easy to pass”
- “That it’s actually not hard to understand”
- “For experienced students a faster pace would be nice”
- “The exercises are not too difficult, and can be completed based on [...] the lectures.”
- “The course is very easy to follow. In some other courses, I find it hard to follow the lectures and understand the material, but it is very simple and fun here.”
- “Attendance not being required, because I have already studied programming in high school, and even had my final exam out of C++ stuff. So now, I can just not attend this class, but still ace the assignments”

Course Difficulty (2/5)

On the other hand:

- “The difficult exercises”
- “intensity of the course”
- “How fast-paced is everything”
- “the complexity increases fast”
- “Sometimes the assignments feel a bit too challenging for someone who started coding 7 weeks ago”
- “It is very intimidating to be in a class where so many people are already experienced in programming.”
- “the fact that the difficulty of the exercises increase way too fast, not giving time for begginers to catch up”
- “The into weeks were difficult, I think I spent more time on google than reading the text book to figure out Java and Intellij”

Course Difficulty (3/5)



My plan: Keep the current level of difficulty, maybe slightly decrease it.

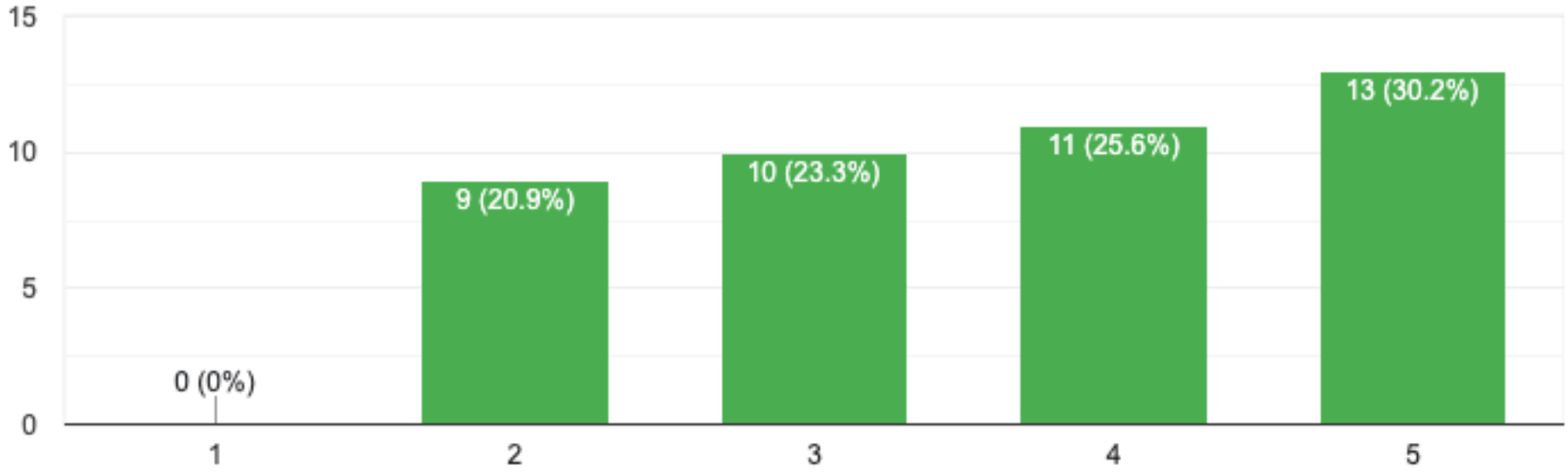
"[...] I think the course is moving way too slowly for people with experience, but it is hard to satisfy everyone, as many with no prior experience find it too fast. I think the separate courses would be a brilliant idea."

Split the Course?

In the future (i.e. next fall), it would be better if the course was split into two: one for beginners and one for experienced programmers

Copy chart

43 responses



Good Things

The best part of the course is:

1. Live programming
2. QA during lectures
3. Quality of exercises and hand-ins
4. Use of real-world tools (Intellij IDEA)
5. Funny jokes and memes

Bad Things

The worst part of the course is:

1. Lectures at 08.15 AM
2. Difficulty is too high

What do you think about using Java for the course?:

What do you think about using Java for the course?:

You mostly said:

“I think it’s ok”

Going Forward

No major changes:

- Keep doing live programming.
- Keep doing QAs during lectures.
- Keep the difficulty the same (or slightly lower).

Sources for images and slides

- https://introcs.cs.princeton.edu/java/lectures/
- https://commons.wikimedia.org/wiki/File:Screenshot_Recursion_via_vlc.png
- https://commons.wikimedia.org/wiki/File:Recursion6.gif
- https://commons.wikimedia.org/wiki/File:Jl_lejaxhe_portrait.jpg
- https://commons.wikimedia.org/wiki/File:%D0%97%D0%B5%D1%80%D0%BA%D0%B0%D0%BB%D0%BE_%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F_01.jpg
- https://en.wikipedia.org/wiki/File:Mandel_zoom_14_satellite_julia_island.jpg
- https://commons.wikimedia.org/wiki/File:Sierpinski_carpet_6.svg