Lecture 8-1

Recursion

<u>How many students in the line?</u>

For each student except the last one, the answer is 1 + the number of students behind him/her. For the last student, the answer is 1.

Recursion

<u>Recursive function</u>: A function that operates on an input by calling itself to operate on a smaller version of that input

Example: $n! = n * (n-1)!$

$4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1$

<u>Recursive procedure:</u> A procedure that solves a problem by applying itself to solve a smaller subset of that problem

Example: List the files and folders in a given folder

<u>Recursive data structure:</u> A data structure consisting of smaller parts that are the same data structure

Example: The string `"abcd"` can be viewed as the character `'a'`, followed by the string `"bcd"`

<u>Recursion</u>

- An elegant way for modeling such cases
- A fundamental CS concept and technique.

Example: factorial

$$factorial(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot factorial(n-1) & \text{otherwise} \end{cases}$$

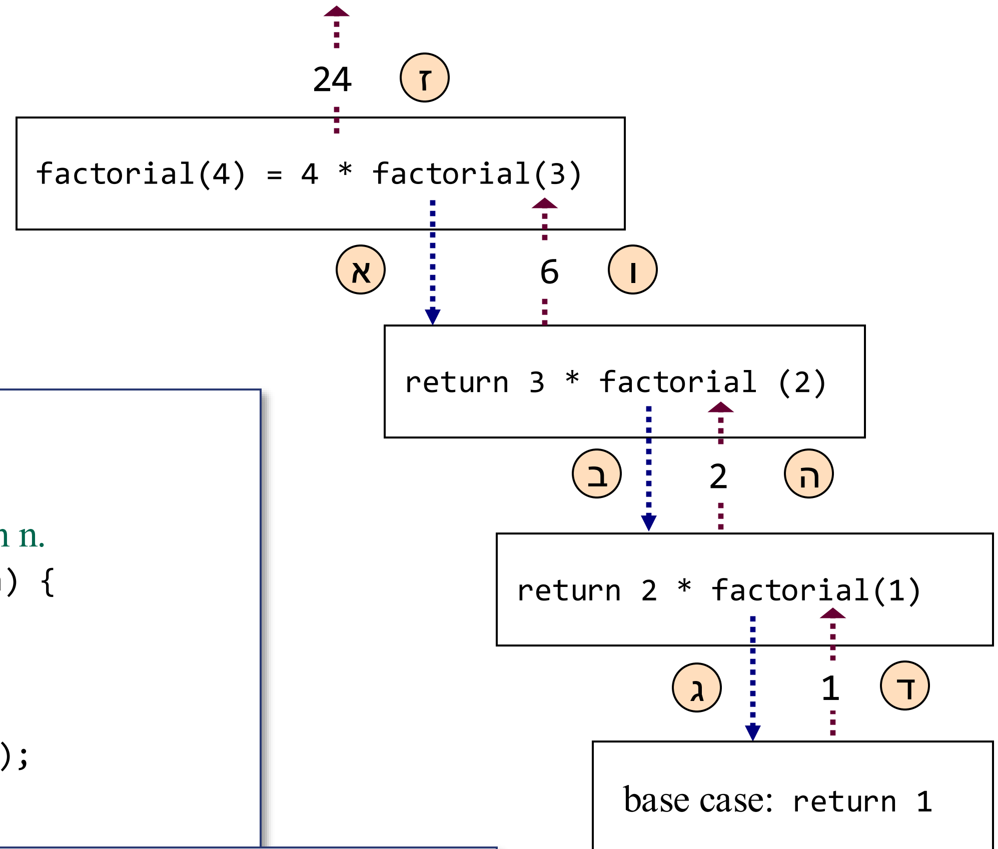
```
factorial(5) = 5 * factorial(4) =
               5 * (4 * factorial(3)) =
               5 *  4 * (3 * factorial(2)) =
               5 *  4 *  3 * (2 * factorial(1)) =
               5 *  4 *  3 *  2 * (1 * 1) =
               5 *  4 *  3 *  2 *  1 = 120
```

```
public class MyMath {
...
    // Returns the factorial (n!) of the given n.
    public static int factorial(int n) {
        if (n == 1)
            return 1;
        else
            return n * factorial(n-1);
    }
}
```

```
public class Foo
   ...
   System.out.println("4! = " +  MyMath.factorial(4));
   System.out.println("10! = " + MyMath.factorial(10));
   ...
```

```
% java Foo
4! = 24
10! = 3628800
```

Example: factorial (simulation)




```
public class MyMath {
...
    // Returns the factorial (n!) of the given n.
    public static int factorial(int n) {
        if (n == 1)
            return 1;
        else
            return n * factorial(n-1);
    }
}
```

```
public class Foo
   ...
   System.out.println("4! = " +  MyMath.factorial(4));
   System.out.println("10! = " + MyMath.factorial(10));
   ...
```

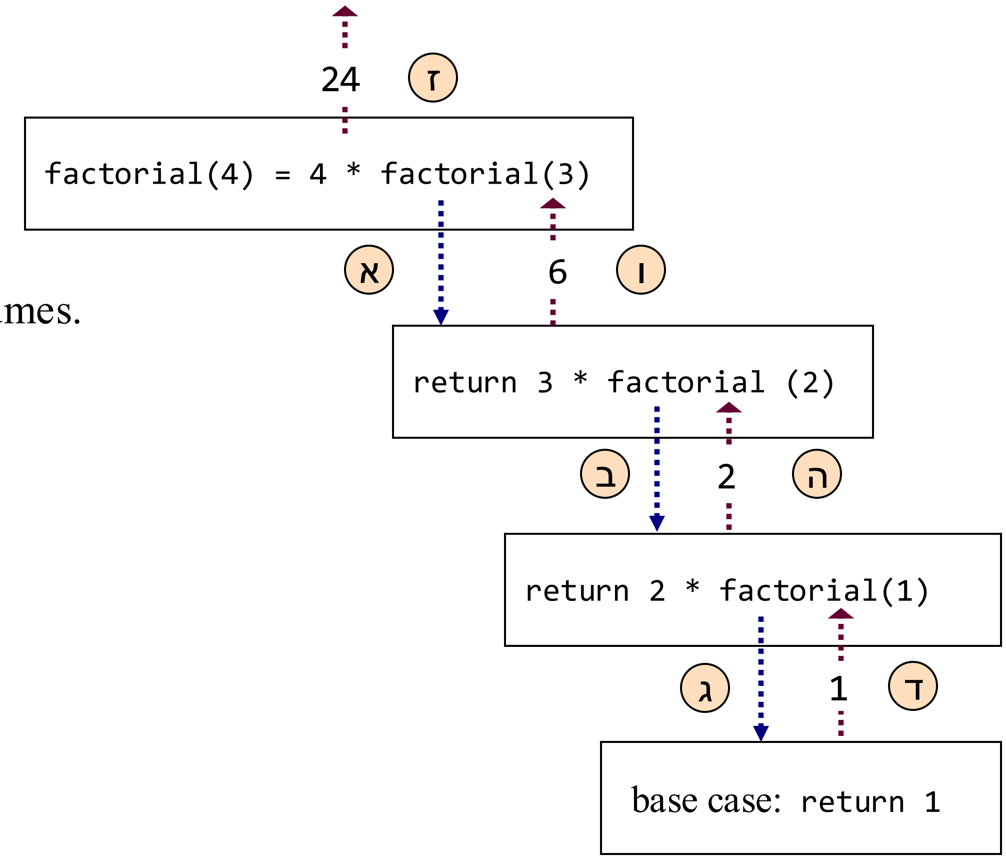
```
% java Foo
4! = 24
10! = 3628800
```

Recursive function calls

<u>Function calling</u> (in general)

When function *f* calls function *g*,
the caller's variables (*parameters*
and *local variables*) are put on hold;

When *g* returns, the caller's variables
are re-instantiated, and its execution resumes.

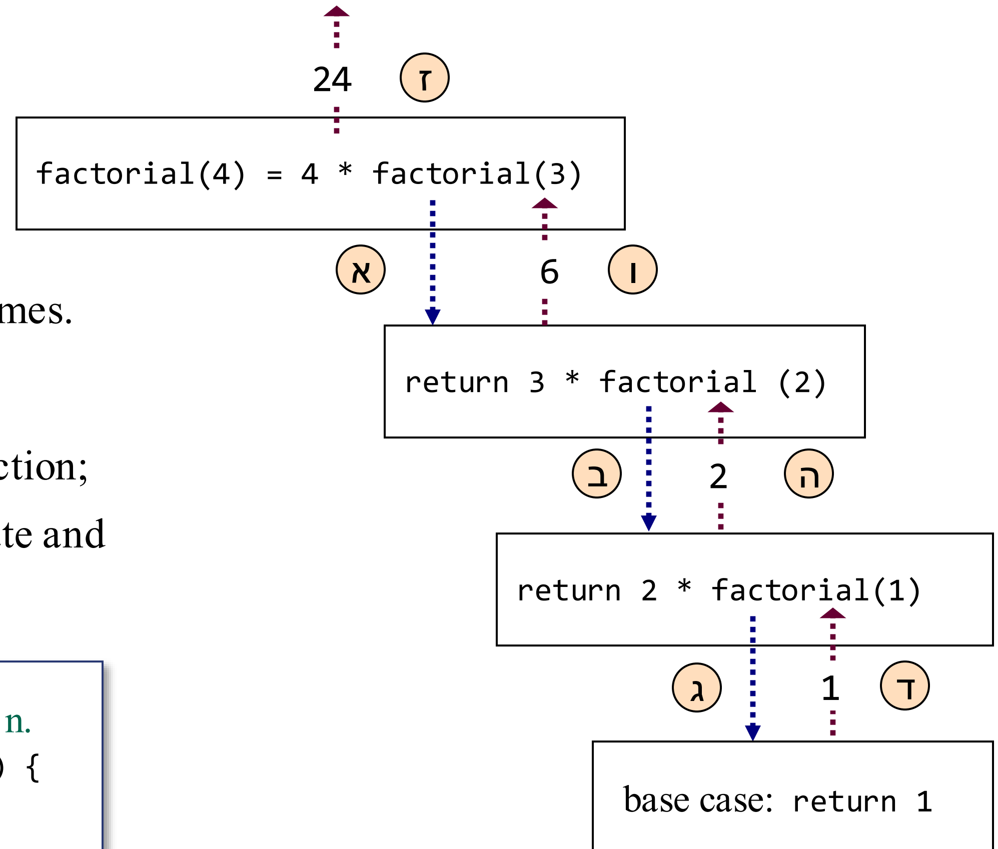


Recursive function calls

<u>Function calling</u> (in general)

When function *f* calls function *g*,
the caller's variables (*parameters*
and *local variables*) are put on hold;

When *g* returns, the caller's variables
are re-instantiated, and its execution resumes.



<u>Recursive function call:</u>

f and *g* are two instances of the same function;

Each recursive call is handled as a separate and
independent function call.

```
// Returns the factorial (n!) of the given n.
public static int factorial(int n) {
   if (n == 1)
      return 1;
   else
      return n * factorial(n-1);
```

variable n is
put on hold

Recursive and iterative implementations

$$factorial(n) = \begin{cases} 1 & if\ n = 1 \\ n \cdot factorial(n-1) & otherwise \end{cases}$$

```
// Returns the factorial (n!) of the given n.
public static int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial((n - 1));
    }
}
```

recursive implementation

$$factorial(n) = \begin{cases} 1 & if\ n = 1 \\ n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 & otherwise \end{cases}$$

```
// Returns the factorial (n!) of a given n.
public static long factorial (int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) {
        fact *= i;
    }
    return fact;
}
```

iterative implementation

- Every recursive implementation has an equivalent iterative implementation
- The recursive implementation is often more elegant
- The iterative implementation looks more efficient
- But, good compilers generate low-level code that converts recursive implementations into iterative implementations.

Recursive intuition

$$factorial(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot factorial(n-1) & \text{otherwise} \end{cases}$$

```
// Returns the factorial (n!) of the given n.
public static int factorial(int n) {
    if (n == 1)
        return 1;
    else
        return n * factorial((n - 1));
    }
}
```

base case

combination

reduction

Recursive algorithms are
based on three elements:

<u>Reduction</u>

Reduce the original problem into a smaller / simpler sub-problem

<u>Base case</u>

The reduction must end with a sub-problem that can be solved directly

<u>Combination</u>

Combining the current "level" with results returned by the level below.

Lecture plan

Recursive functions (examples)

- Factorial
- String processing
- Fibonacci
- Power

Recursive procedures (examples)

- Printing
- Fractals
- Permutations

Reversing a string

<u>Task:</u> reverse a string

Example: `reverse("abcd")` should return `"dcba"`

Insight: `reverse("abcd") = reverse("bcd") + ‘a’`

`reverse("a") = "a"`

Reversing a string

Task: reverse a string

Example: `reverse("abcd")` should return `"dcba"`

Insight: `reverse("abcd") = reverse("bcd") + 'a'`

`reverse("a") = "a"`

Recursive algorithm

If `str` consists of a single character, or `null`, return `str`

else return `reverse` (the last $n-1$ characters of `str`) + `str[0]`

```
public static String reverse(String str) {
   if (str.length() <= 1) {
      return str;
   }
   return reverse(str.substring(1)) + str.charAt(0));
}
```

base case

reduction

combination

`substring(1)`:

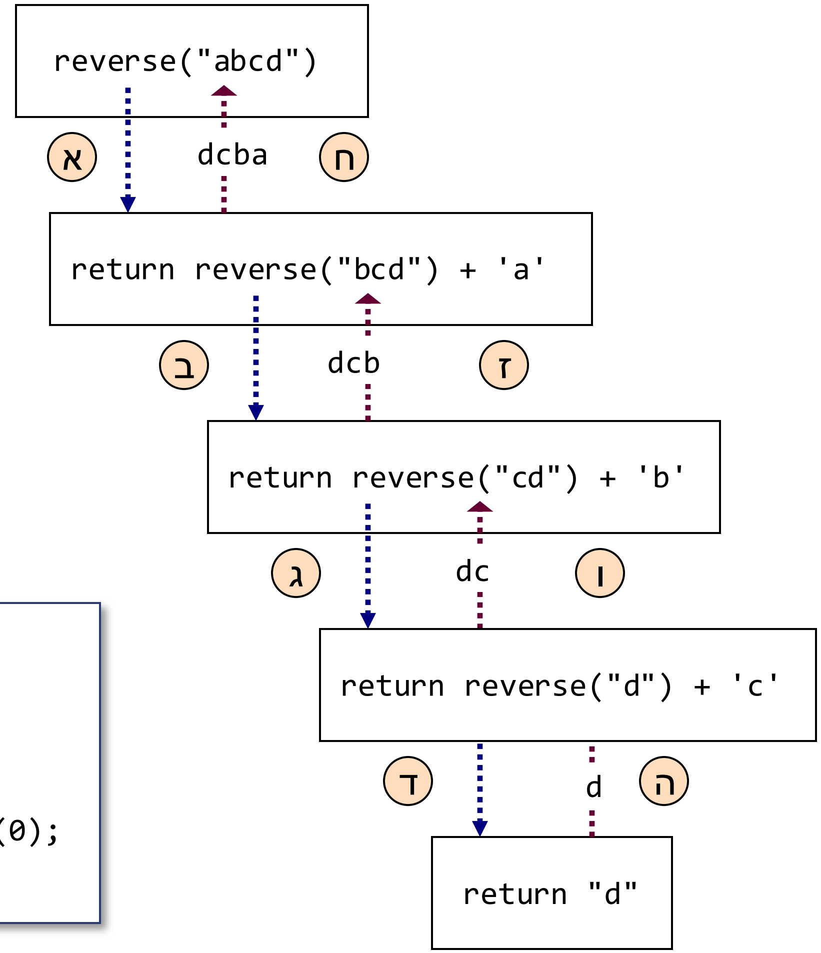
All the characters except the first one

Reversing a string: Simulation

Typical recursive execution pattern

The problem is reduced “on the way down”
(by the recursive calls)

The solution is assembled “on the way up”
(using the return values).




```
public static String reverse(String str) {
   if (str.length() <= 1) {
      return str;
   }
   return reverse(str.substring(1)) + str.charAt(0);
}
```

Lecture plan

Recursive functions (examples)

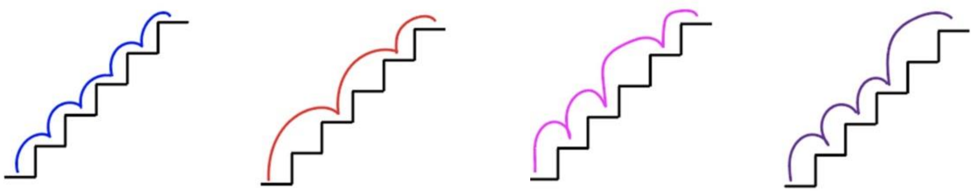
- Factorial
- String processing
- Fibonacci
- Power

Recursive procedures (examples)

- Printing
- Fractals
- Permutations

Fibonacci

When climbing a staircase, you are allowed to climb either 1 stair, or 2 stairs, in each step:



There are many different ways to climb n stairs (using 1 or 2 stairs in each step);

How many ways?

$f(n)$ = number of *different ways* to reach stair n

<u>Insight:</u>

You reach step n either from stair $n-1$, or from stair $n-2$; Therefore:

$$f(n) = f(n-1) + f(n-2)$$

<u>Base cases</u>

$$f(1) = 1$$

$$f(0) = 0$$

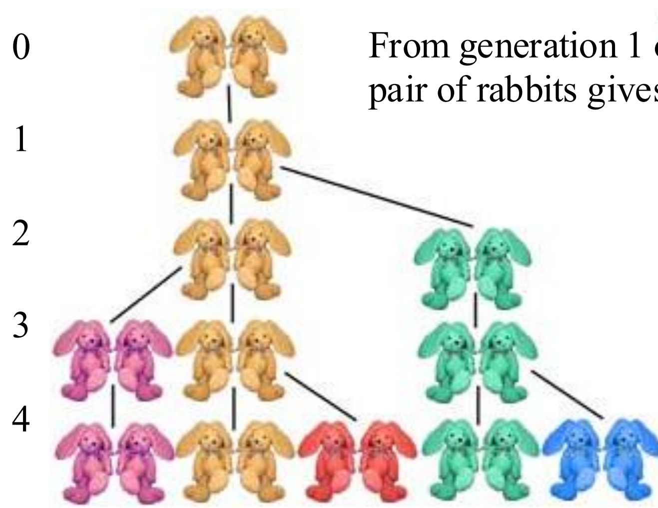
Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \quad \text{for all } n > 1$$

The staircase problem is one out of numerous settings in nature and in engineering in which the Fibonacci series comes up.

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...



From generation 1 onward, in each generation, each pair of rabbits gives birth to a pair of rabbits.

$f(n)$ = number of pairs of rabbits in generation n

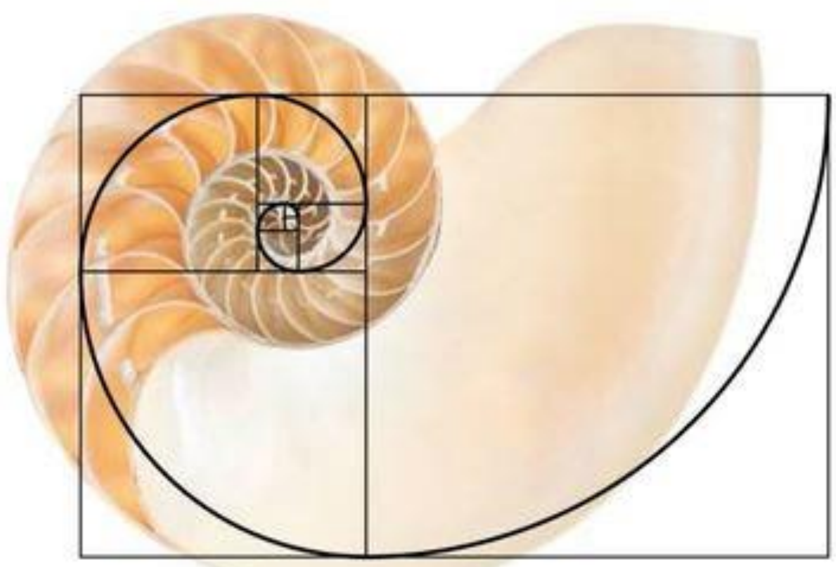
Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \quad \text{for all } n > 1$$

The staircase problem is one out of numerous settings in nature and in engineering in which the Fibonacci series comes up.

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...



$f(n)$ = square length in stage n

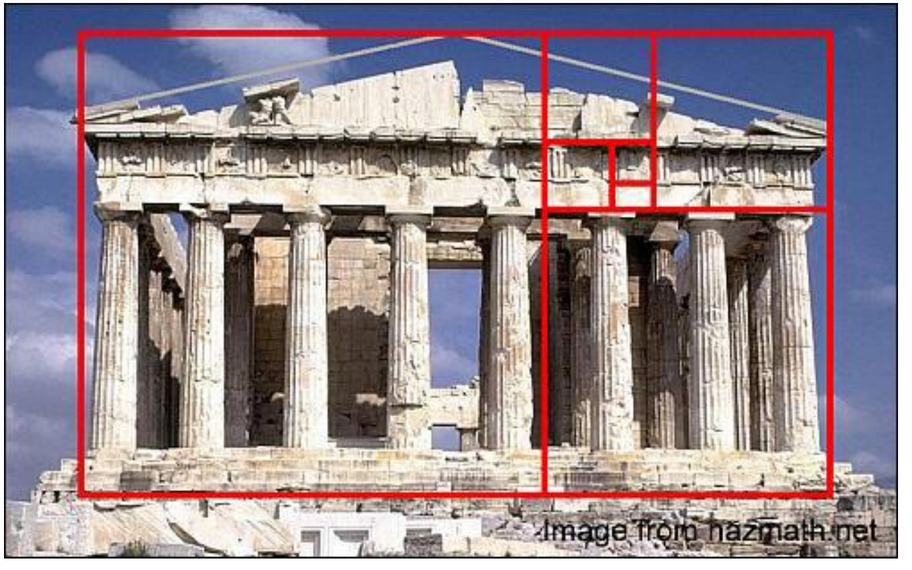
Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \quad \text{for all } n > 1$$

The staircase problem is one out of numerous settings in nature and in engineering in which the Fibonacci series comes up.

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...



$f(n)$ = square length in stage n

Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \quad \text{for all } n > 1$$

The staircase problem is one out of numerous settings in nature and in engineering in which the Fibonacci series comes up.

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...

Number of petals in flowers = Fibonacci numbers

Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \quad \text{for all } n > 1$$

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...

Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \quad \text{for all } n > 1$$

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
// Returns the n'th Fibonacci number
public static int fibonacci1(int n) {
    if (n <= 1) return n;
    int f, fprev = 1, fprevprev = 1;
    for (int i = 2; i <= n; i++) {
        f = fprev + fprevprev;
        fprevprev = fprev;
        fprev = f;
    }
    return f;
}
```

Iterative implementation

```
public class Foo
  ...
  System.out.println(MyMath.fibonacci1(40));
  ...
```

```
% java Foo
102334155
```

Fibonacci

Fibonacci series definition

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \qquad \text{for all } n > 1$$

The Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
// Returns the n'th fibonacci number
public static int fibonacci(int n) {
   if (n <= 1) return n;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursive implementation

```
public class Foo
   ...
   System.out.println(MyMath.fibonacci(40));
   ...
```

```
% java Foo
102334155
```

Performance issues

Recursive solution

```
// Returns the n'th fibonacci number
public static int fibonacci(int n) {
   if (n <= 1) return n;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

Iterative solution

```
// Returns the n'th fibonacci number
public static int fibonacci1(int n) {
    if (n <= 1) return n;
    int f, fprev = 1, fprevprev = 1;
    for (int i = 2; i <= n; i++) {
        f = fprev + fprevprev;
        fprevprev = fprev;
        fprev = f;
    }
    return f;
}
```

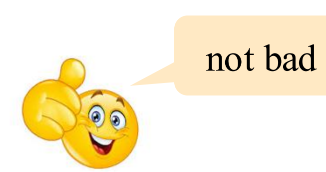
<u>Running time</u>

The computation of `fibonacci(`n`)` requires... Let's see...

<u>Running time</u>

The computation of `fibonacci1(`n`)` requires n steps

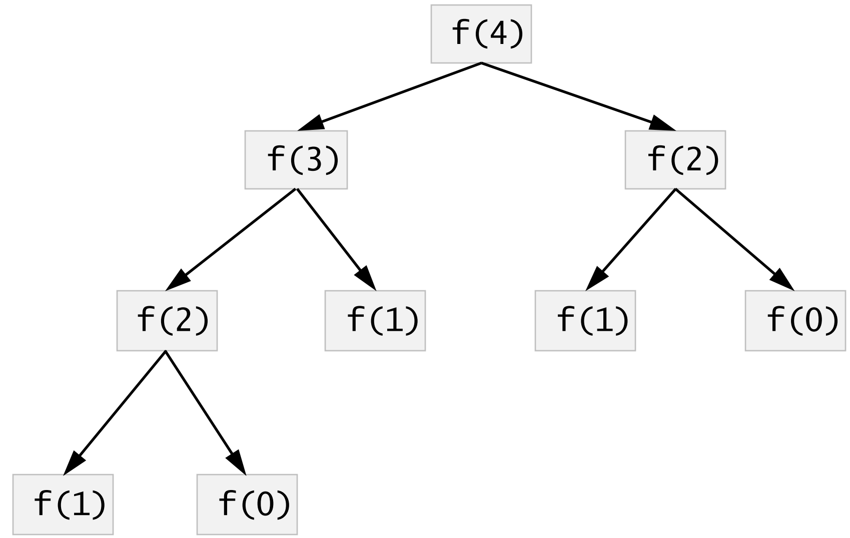
(***linear running time***)



Performance issues

Recursive solution

```
// Returns the n'th fibonacci number
public static int fibonacci(int n) {
   if (n <= 1) return n;
   return fibonacci(n-1) + fibonacci(n-2);
}
```



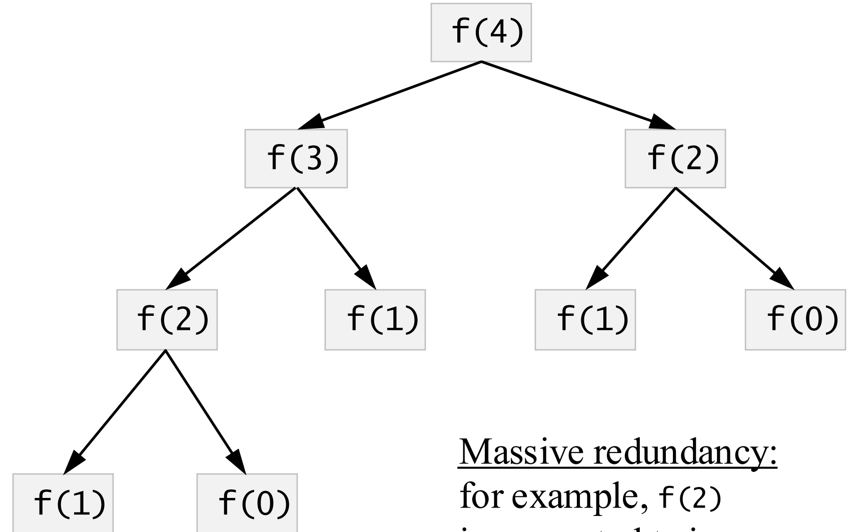
<u>Running time</u>

The computation of `fibonacci(`n`)` requires... Let’s see...

Performance issues

Recursive solution

```
// Returns the n'th fibonacci number
public static int fibonacci(int n) {
   if (n <= 1) return n;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

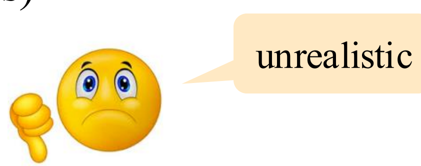


<u>Massive redundancy:</u>
for example, `f(2)`
is computed twice

<u>Running time</u>

The computation of `fibonacci(`n`)`
requires 2^n steps (recursive calls)

(***exponential* running time**)



<u>Optimizing recursive solutions:</u>

Memoization: After computing $f(n)$, store it, for re-use.

Lecture plan

<u>Recursive functions</u> (examples)

- Factorial
- String processing
- Fibonacci
- Power

<u>Recursive procedures</u> (examples)

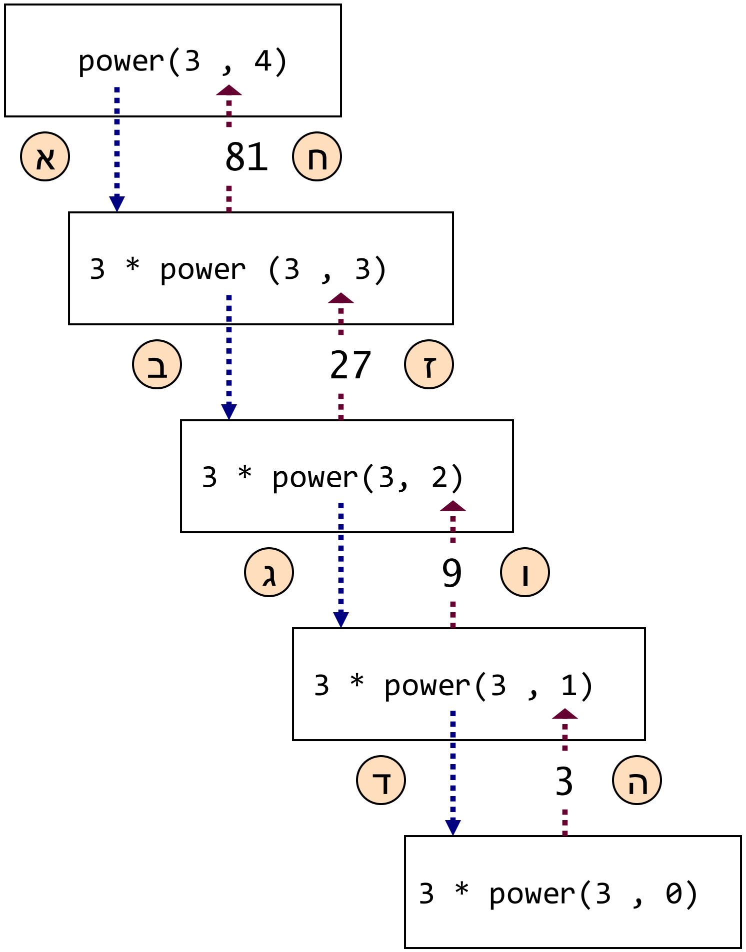
- Printing
- Fractals
- Permutations

Power: x^n

$$\text{power}\ (x\ ,\ 0) = 1$$

$$\text{power}\ (x\ ,\ n) = x\ *\ \text{power}\ (x\ ,\ n-1) \quad \text{for } n > 0$$

```
// Returns x raised to the power of n
public static int power(int x, int n) {
    if (n == 0) return 1;
    return x * power(x, n - 1);
}
```



Running time

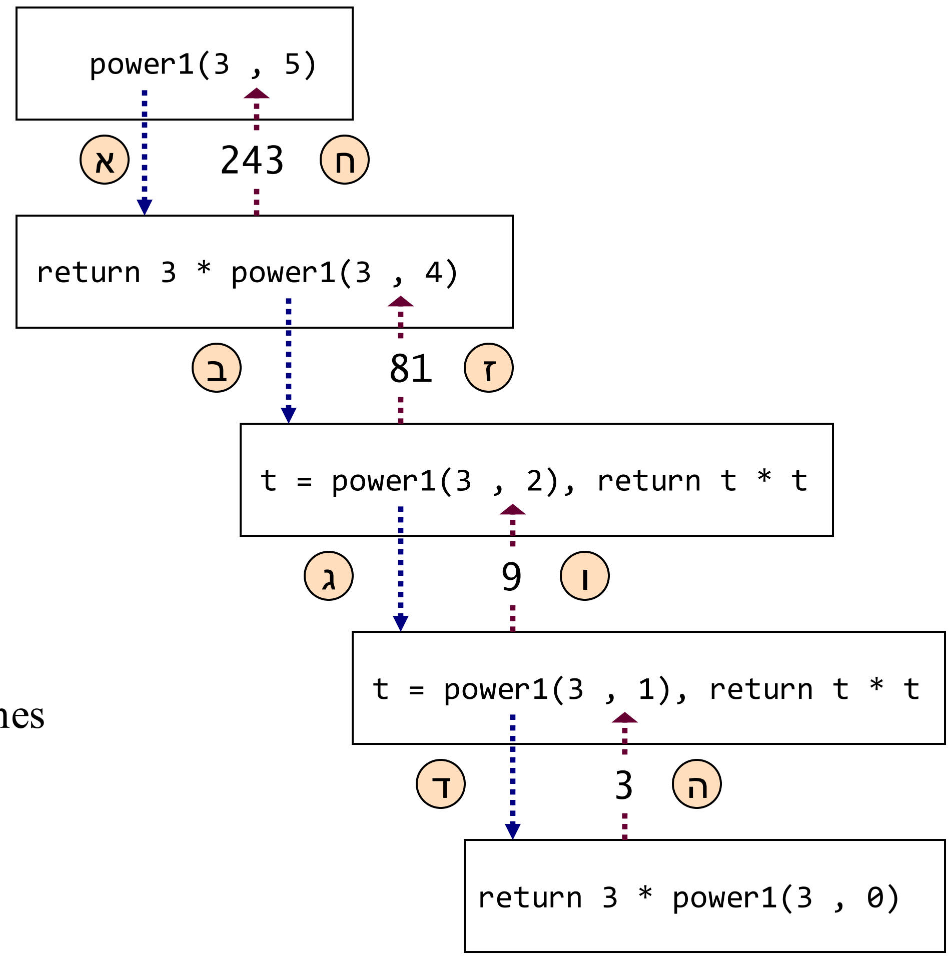
The computation of `power(x,n)` requires `n` steps (recursive calls)

(***linear* running time**)

Can we do better?

Power: $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$

```
// Returns x raised to the power of n
public static int power1(int x, int n) {
    if (n==0) return 1;
    if ((n%2)==0) {
        int t = power1(x,n/2);
        return t*t;
    }
    return x * power1(x,n-1);
}
```



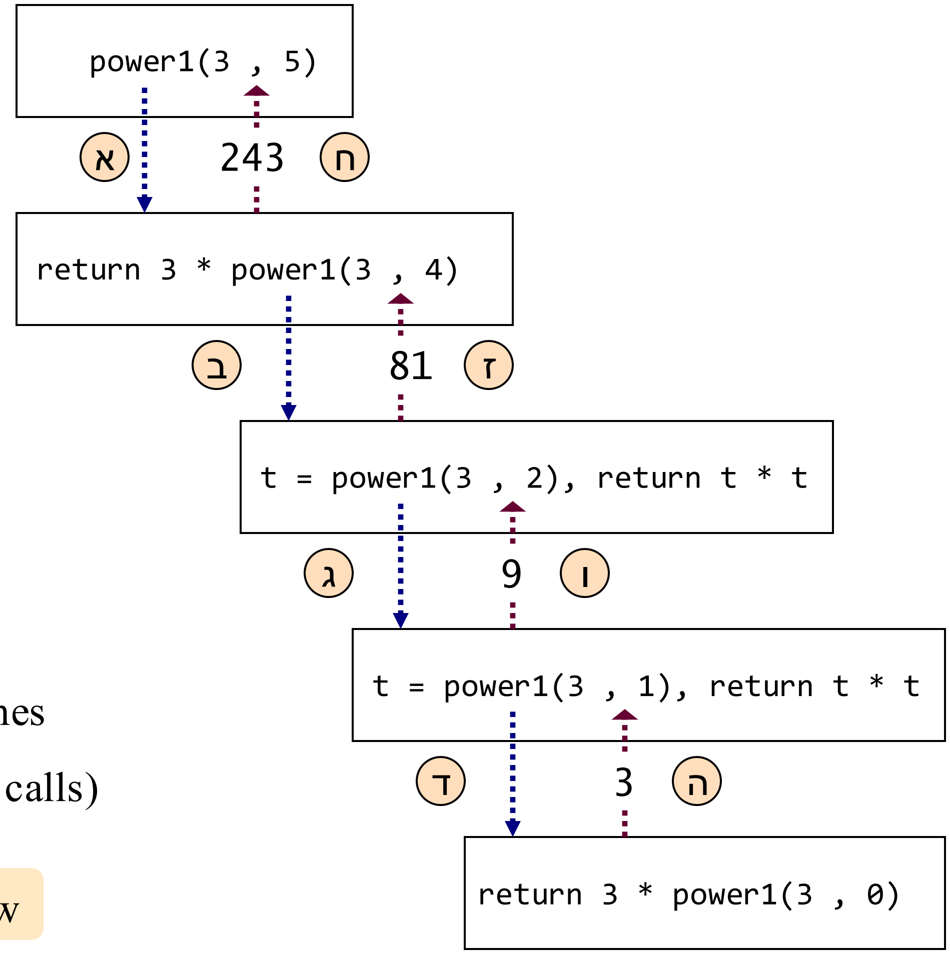
<u>Running-time</u>

In each step we either call `power1(x,n/2)`,
or we call `power1(x,n-1)`

How many times can n be divided by 2? $\log_2 n$ times

Power: $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$

```
// Returns x raised to the power of n
public static int power1(int x, int n) {
    if (n==0) return 1;
    if ((n%2)==0) {
        int t = power1(x,n/2);
        return t*t;
    }
    return x * power1(x,n-1);
}
```



<u>Running-time</u>

In each step we either call `power1(x,n/2)`,
or we call `power1(x,n-1)`

How many times can n be divided by 2? $\log_2 n$ times

The running time is at most $\log_2 n$ steps (recursive calls)

(*logarithmic* running time
is dramatically faster
than *linear* running time)

How can we “trust an algorithm” (in general)

```
// Returns x raised to the power of n
public static int power1(int x, int n) {
    if (n==0) return 1;
    if ((n%2)==0) {
        int t = power1(x,n/2);
        return t*t;
    }
    return x * power1(x,n-1);
}
```

Two approaches

- Test the algorithm (empirical)
- Prove that it works (formal)

Theorem: For any x and positive integer n, this algorithm returns x^n

Proof: (by induction)

Base case: if $n = 0$ the algorithm returns 1.

Inductive hypothesis: assume that for all $k < n$ the algorithm returns x^k

Inductive step:

If n is even, the algorithm returns power(x, n/2) * power(x, n/2);

By the induction hypothesis, power(x, n/2) returns $x^{½ n}$.

Therefore, the algorithm returns $(x^{½ n}) * (x^{½ n}) = x^n$

If n is odd, the algorithm returns x * power (x, n-1);

By the induction hypothesis, power (x, n-1) returns x^{n-1}.

Therefore, the algorithm returns $x * (x^{n-1}) = x^n$.

Lecture plan

<u>Recursive functions</u> (examples)

- Factorial
- String processing
- Fibonacci
- Power

<u>Recursive procedures</u> (examples)

- Printing
- Fractals
- Permutations

Print, reversed

Print reverse:

```
// Inputs numbers from the user;
// When 0 is entered, prints the numbers, in reverse
private static void printReverse() {
```

```
% java PrintReverse
Enter a number: 5
Enter a number: 2
Enter a number: 7
Enter a number: 0
7
2
5
```

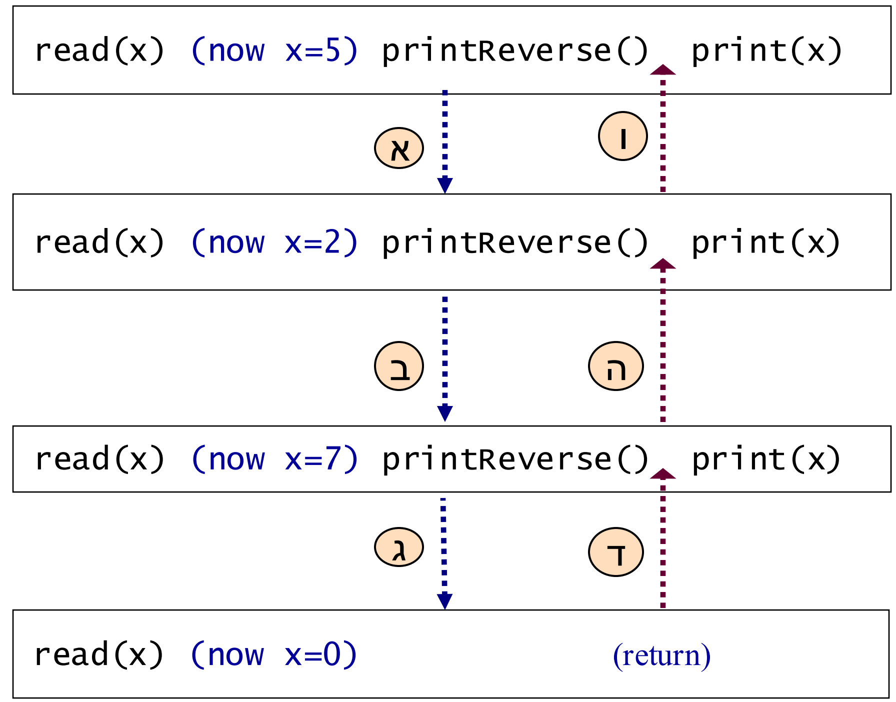
Print, reversed

Print reverse:

```
// Inputs numbers from the user;
// When 0 is entered, prints the numbers, in reverse
private static void printReverse() {
    In in = new In();
    System.out.print("Enter a number: ");
    int x = in.readInt();
    if (x != 0) {
        printReverse();
        System.out.println(x);
    }
}
```

```
% java PrintReverse
Enter a number: 5
Enter a number: 2
Enter a number: 7
Enter a number: 0
7
2
5
```

Suppose that the user inputs: 5, 2, 7, 0



In each function call:
The local variable x is put "on hold"

When we return from a call:
We revisit the x values, and print them

Print, reversed

Print reverse:

```
// Inputs numbers from the user;
// When 0 is entered, prints the numbers, in reverse
private static void printReverse() {
   In in = new In();
   System.out.print("Enter a number: ");
   int x = in.readInt();
   if (x != 0) {
      printReverse();
      System.out.println(x);
   }
}
```

Print reverse, reversed:

```
// Read and print numbers,
// until 0 is entered
private static void printReverse1() {
   In in = new In();
   System.out.print("Enter a number: ");
   int x = in.readInt();
   if (x != 0) {
      System.out.println(x);
      printReverse1();
   }
}
```

```
Enter a number: 5
5
Enter a number: 2
2
Enter a number: 7
7
Enter a number: 0
```

- Recursive version of a while loop
- In principle, every loop can be implemented recursively

Lecture plan

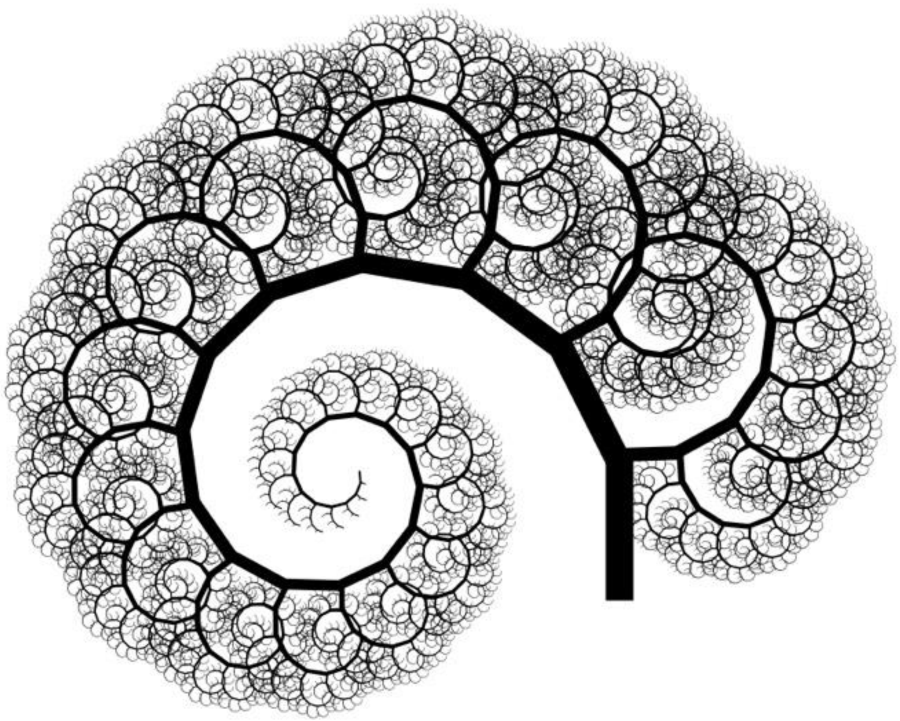
Recursive functions (examples)

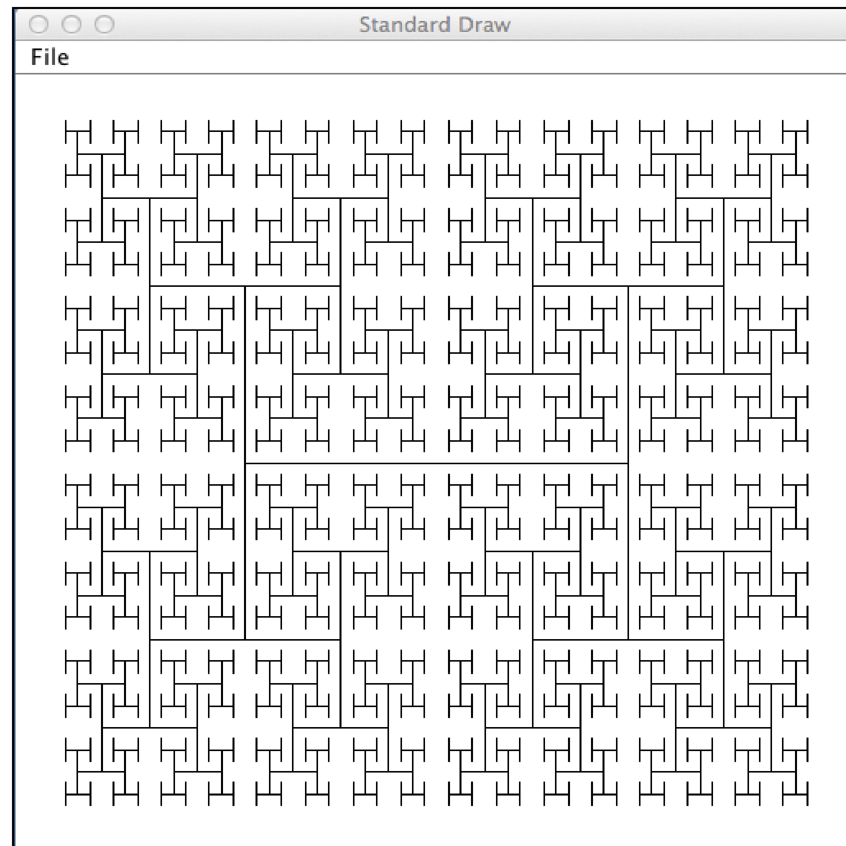
- Factorial
- String processing
- Fibonacci
- Power

Recursive procedures (examples)

- Printing
- Fractals
- Permutations

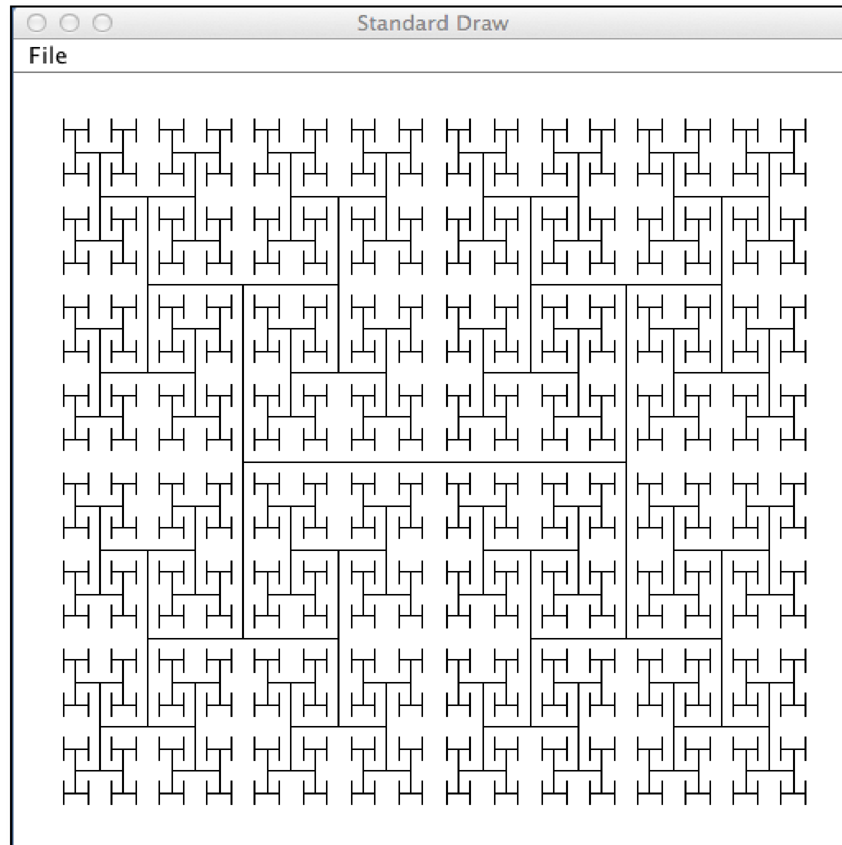
Fractal drawing





Fractal drawing

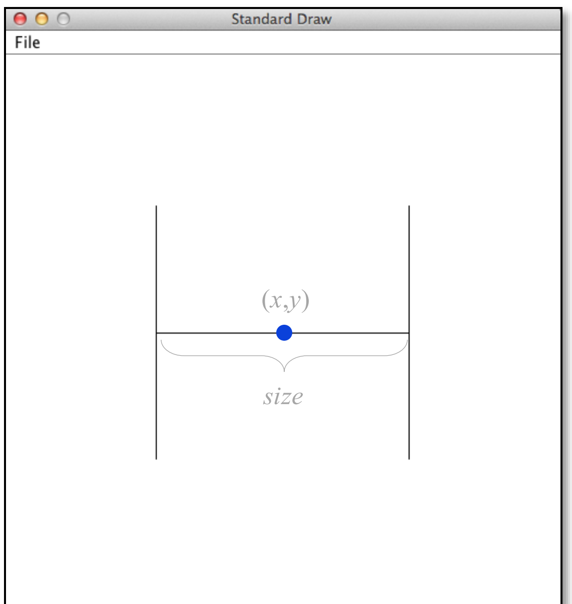
Task: draw a "fractal H figure"

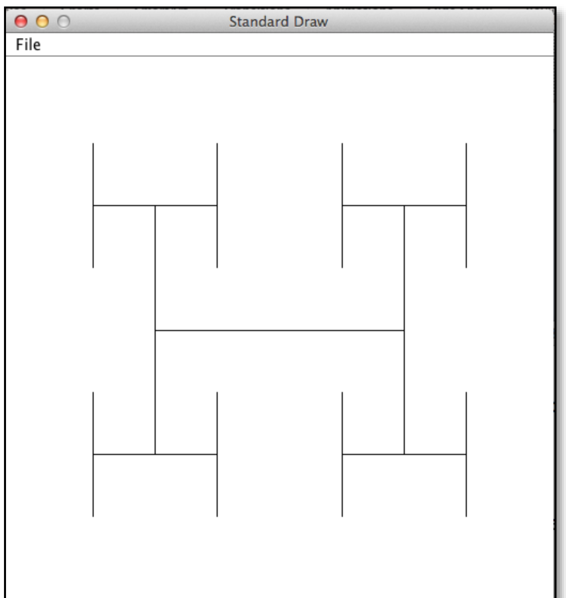


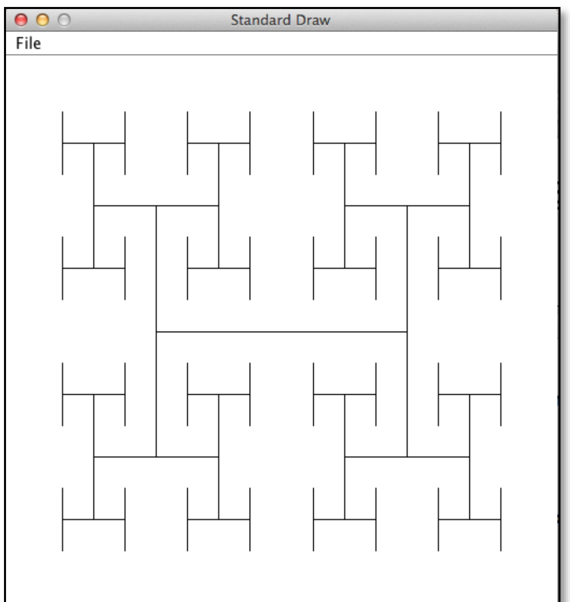
Fractal drawing

<u>Task:</u> draw a "fractal H figure"

- Draw an H figure of a given *size*, centered at (*x*, *y*)
- Draw 4 H figures of *half the size*, centered at the 4 tips of the H
- Draw 4 H figures of *half the size*, centered at the 4 tips of every H
- Etc.





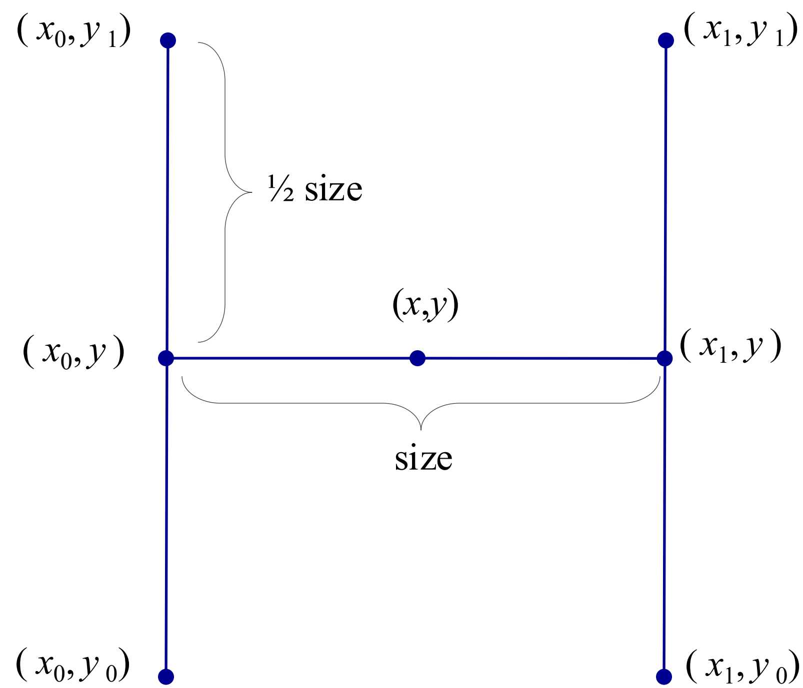


Fractal drawing

<u>Task:</u> draw a "fractal `H` figure"

- Draw an `H` figure of a given *size*, centered at (x, y)
- Draw 4 `H` figures of *half the size*, centered at the 4 tips of the H
- Draw 4 `H` figures of *half the size*, centered at the 4 tips of every H
- Etc.

```
public static void drawH(double x, double y,
                         double size) {
```

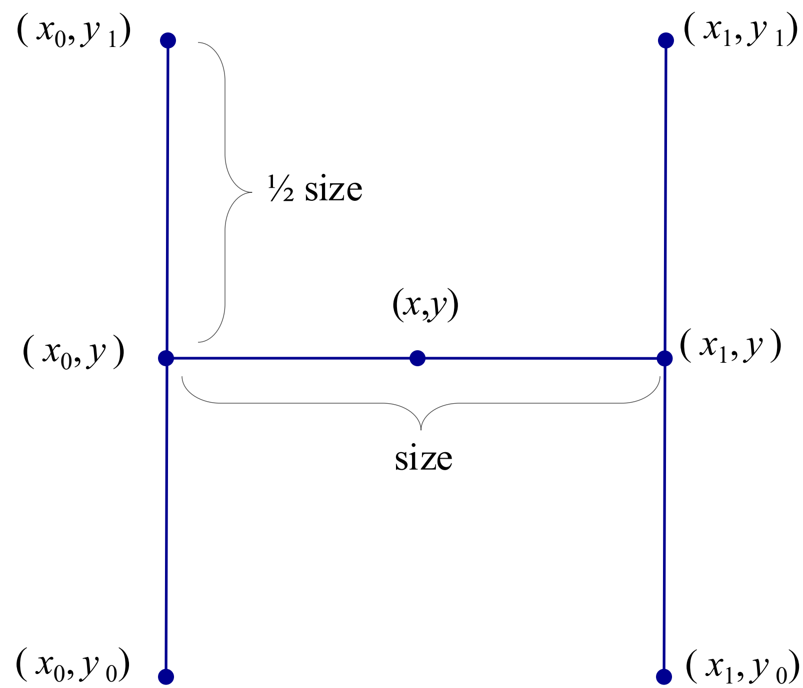


Fractal drawing

Task: draw a "fractal H figure"

- Draw an H figure of a given *size*, centered at (x, y)
- Draw 4 H figures of *half the size*, centered at the 4 tips of the H
- Draw 4 H figures of *half the size*, centered at the 4 tips of every H
- Etc.

```
public static void drawH(double x, double y,
                         double size) {
   double x0 = x - size/2, x1 = x + size/2;
   double y0 = y - size/2, y1 = y + size/2;
   // Draws the H figure
   StdDraw.line(x0, y, x1, y);
   StdDraw.line(x0, y0, x0, y1);
   StdDraw.line(x1, y0, x1, y1);
   // Draws 4 H figures of half the size, at the
   // four tips of the current H figure
   drawH(x0, y0, size/2);
   drawH(x0, y1, size/2);
   drawH(x1, y0, size/2);
   drawH(x1, y1, size/2);
}
```



Fractal drawing

<u>Task:</u> draw a “fractal `H` figure”

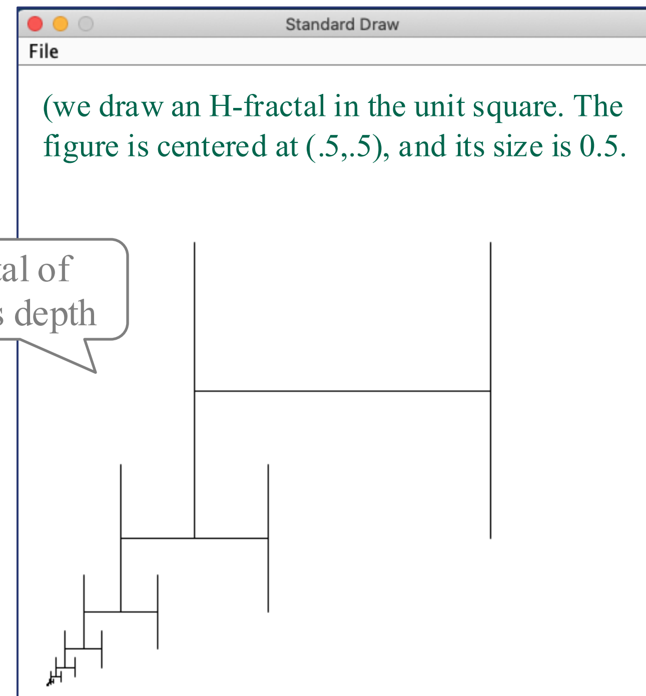
- Draw an `H` figure of a given *size*, centered at (x, y)
- Draw 4 `H` figures of *half the size*, centered at the 4 tips of the H
- Draw 4 `H` figures of *half the size*, centered at the 4 tips of every H
- Etc.

```
public static void drawH(double x, double y,
                         double size) {
   double x0 = x - size/2, x1 = x + size/2;
   double y0 = y - size/2, y1 = y + size/2;
   // Draws the H figure
   StdDraw.line(x0, y, x1, y);
   StdDraw.line(x0, y0, x0, y1);
   StdDraw.line(x1, y0, x1, y1);
   // Draws 4 H figures of half the size, at the
   // four tips of the current H figure
   drawH(x0, y0, size/2);
   drawH(x0, y1, size/2);
   drawH(x1, y0, size/2);
   drawH(x1, y1, size/2);
}
```

<u>Needed:</u> a “base case” that stops the recursion

drawH(.5, .5, .5)

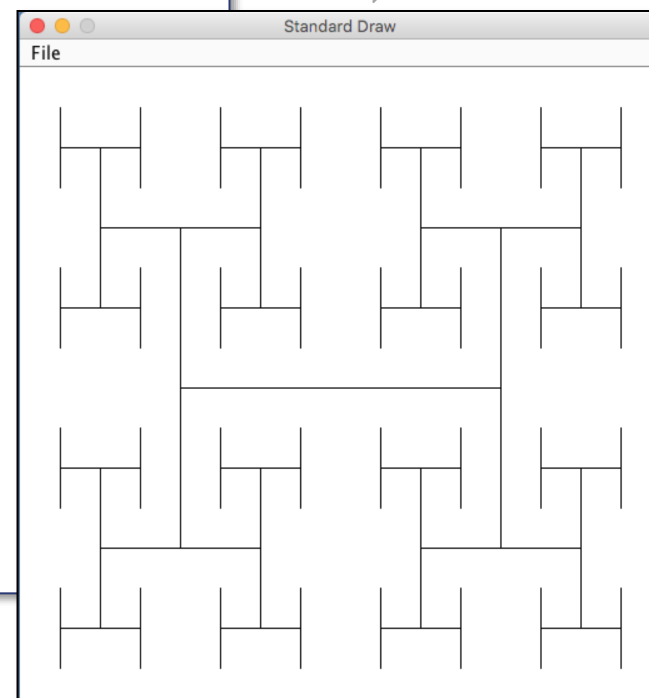
H-fractal of endless depth



Fractal drawing

```
public static void test() {
    // Draws an H-fractal of depth 3, in the unit square.
    // The figure will be centered at (.5,.5), and its size will be 0.5.
    drawH(.5, .5, .5, 3);
}
```

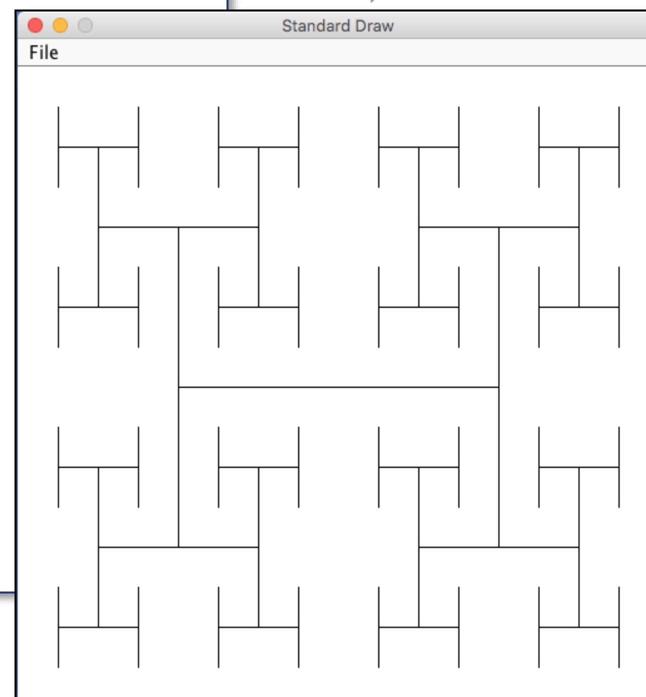
H-fractal of depth n = 3



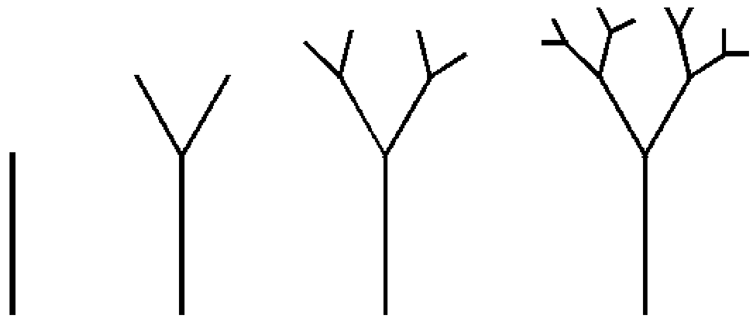
Fractal drawing

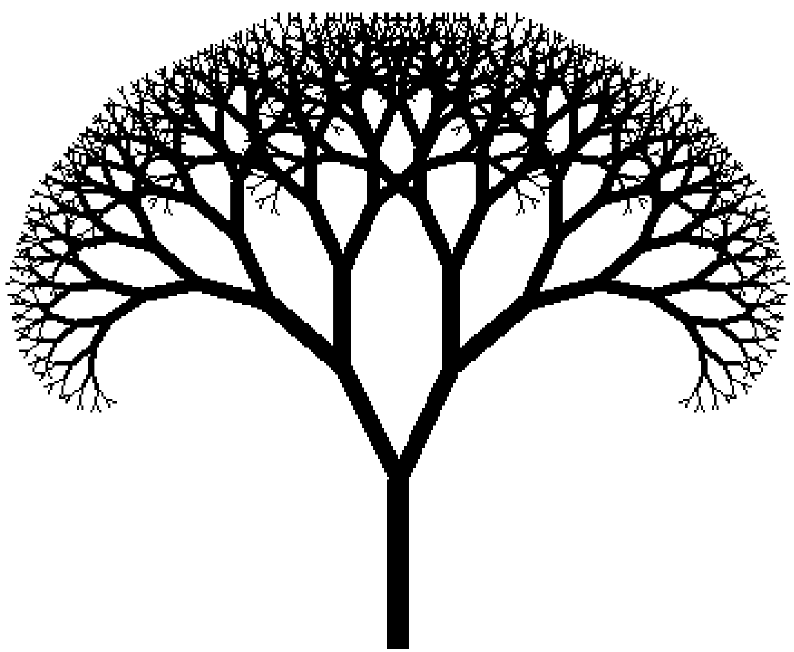
```
public static void test() {
    // Draws an H-fractal of depth 3, in the unit square.
    // The figure will be centered at (.5,.5), and its size will be 0.5.
    drawH(.5, .5, .5, 3);
}

// Draws an H-fractal, centered at x,y, of the given size and depth.
public static void drawH(double x, double y, double size, int n) {
    if (n == 0) return;
    double x0 = x - size/2, x1 = x + size/2;
    double y0 = y - size/2, y1 = y + size/2;
    // Draws the H figure
    StdDraw.line(x0, y, x1, y);
    StdDraw.line(x0, y0, x0, y1);
    StdDraw.line(x1, y0, x1, y1);
    // Draws 4 H figures of half the size,
    // located at the 4 tips of the current H figure
    drawH(x0, y0, size/2, n - 1);
    drawH(x0, y1, size/2, n - 1);
    drawH(x1, y0, size/2, n - 1);
    drawH(x1, y1, size/2, n - 1);
}
```

H-fractal of depth n = 3



Fractal trees





See lecture code:

- `Tree.java`
- `TreeNatural.java`

Fractal trees, with noise

Computer generated landscapes

Lecture plan

<u>Recursive functions</u> (examples)

- Factorial
- String processing
- Fibonacci
- Power

<u>Recursive procedures</u> (examples)

- Printing
- Fractals
- Permutations

Permutations

Recursive insight

- List all the strings that start with `"a"`, followed by all the permutations of `"bcd"`
- List all the strings that start with `"b"`, followed by all the permutations of `"acd"`
- List all the strings that start with `"c"`, followed by all the permutations of `"abd"`
- List all the strings that start with `"d"`, followed by all the permutations of `"abc"`

```
listPerms("abcd"):
```

```
abcd
abdc
acbd
acdb
adbc
adcb
----
bacd
badc
bcad
bcda
bdac
bdca
----
cabd
cadb
cbad
cbda
cdab
cdba
----
dabc
dacb
dbac
dbca
dcab
dcba
```

Permutations

```
// Prints all the permutations of s
public static void listPerms (String s) {
    listPerms("", s);
}

// Prints the given prefix, followed by all the permutations of s
private static void listPerms (String prefix, String s) {
```

Permutations

```
// Prints all the permutations of s
public static void listPerms (String s) {
    listPerms("", s);
}

// Prints the given prefix, followed by all the permutations of s
private static void listPerms (String prefix, String s) {
   if (s.length() == 0)
      System.out.println(prefix);
   else
      for (int i = 0; i < s.length(); i++) {
         // ch = i'th character of the string s
         char ch = s.charAt(i);
         // rest = s minus ch
         String rest = s.substring(0,i) + s.substring(i+1);
         listPerms(prefix + ch, rest);
      }
   }
}
```

listPerms("abc"):

```
calling listPerms(a, bc)
```

```
// Debugging print:
System.out.println("calling listPerms(" + (prefix + ch) + ", " + rest + ")");
```

Permutations

```
// Prints all the permutations of s
public static void listPerms (String s) {
    listPerms("", s);
}

// Prints the given prefix, followed by all the permutations of s
private static void listPerms (String prefix, String s) {
   if (s.length() == 0)
      System.out.println(prefix);
   else
      for (int i = 0; i < s.length(); i++) {
         // ch = i'th character of the string s
         char ch = s.charAt(i);
         // rest = s minus ch
         String rest = s.substring(0,i) + s.substring(i+1);
         listPerms(prefix + ch, rest);
      }
   }
}
```

listPerms("abc"):

```
calling listPerms(a, bc)
calling listPerms(ab, c)
calling listPerms(abc, )
abc
calling listPerms(ac, b)
calling listPerms(acb, )
acb
calling listPerms(b, ac)
calling listPerms(ba, c)
calling listPerms(bac, )
bac
calling listPerms(bc, a)
calling listPerms(bca, )
bca
calling listPerms(c, ab)
calling listPerms(ca, b)
calling listPerms(cab, )
cab
calling listPerms(cb, a)
calling listPerms(cba, )
cba
```

```
// Debugging print:
System.out.println("calling listPerms(" + (prefix + ch) + ", " + rest + ")");
```