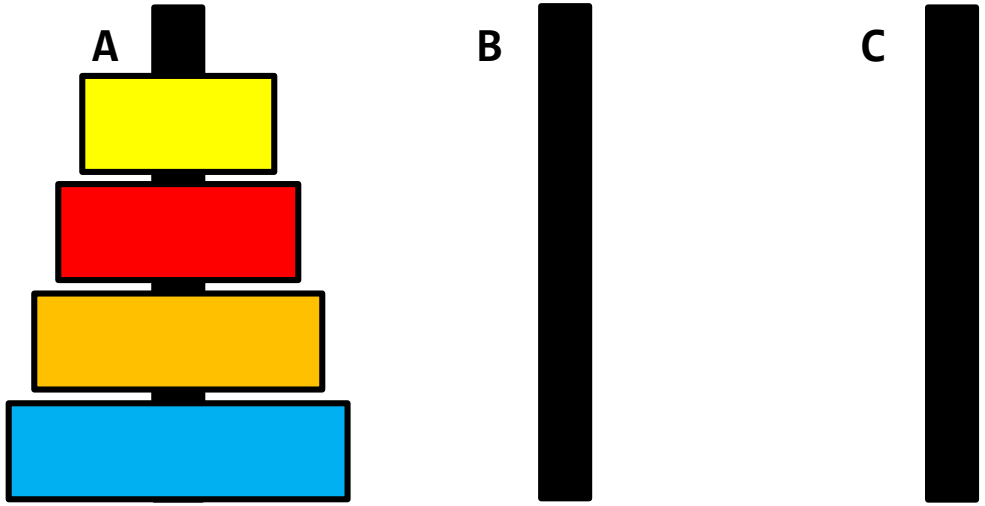
Recursion

notes Chapter 8

Tower of Hanoi



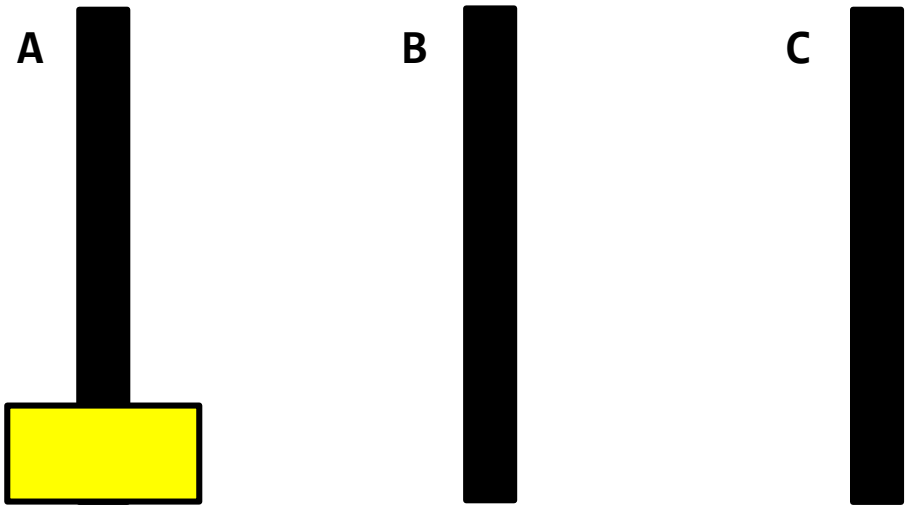
- move the stack of *n* disks from A to C
  - can move one disk at a time from the top of one stack onto another stack
  - cannot move a larger disk onto a smaller disk

Tower of Hanoi

- legend says that the world will end when a 64 disk version of the puzzle is solved
- several appearances in pop culture
  - Doctor Who
  - Rise of the Planet of the Apes
  - Survior: South Pacific

Tower of Hanoi

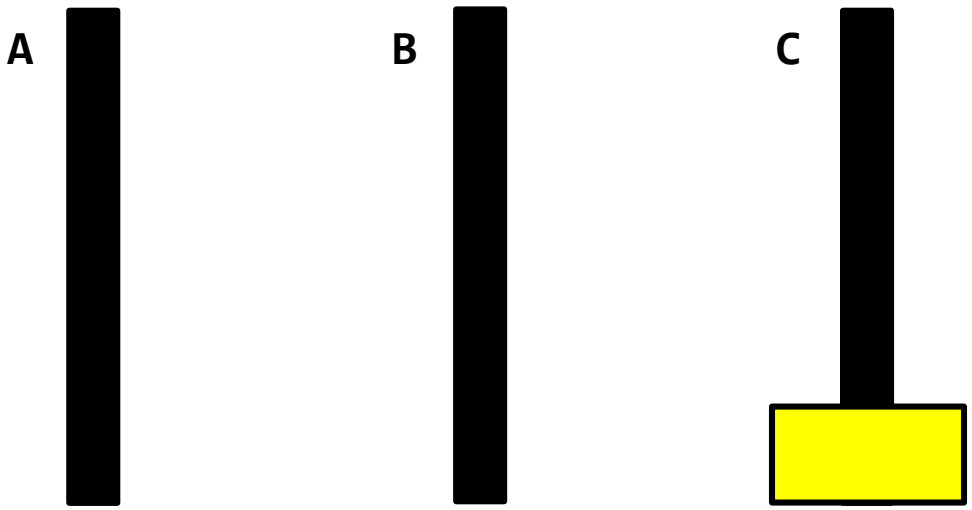
- n = 1



- move disk from A to C

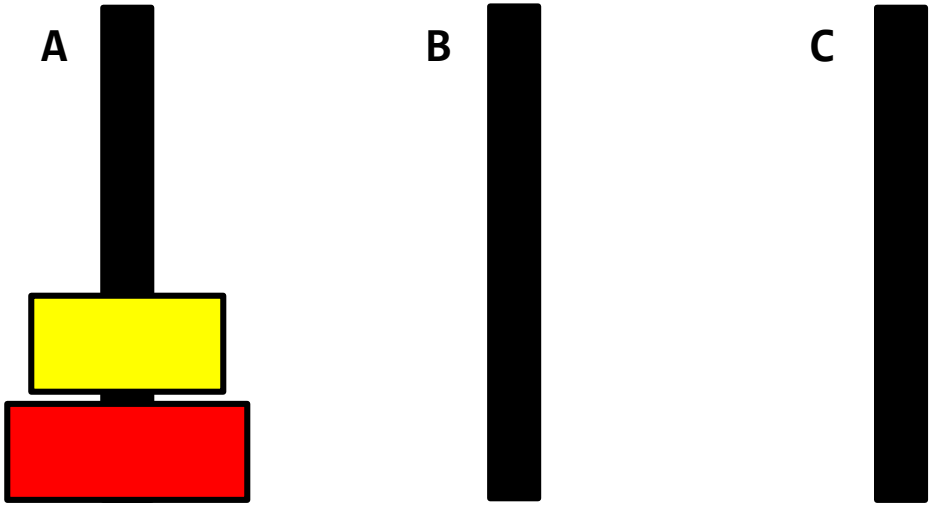
Tower of Hanoi

- $n = 1$



Tower of Hanoi

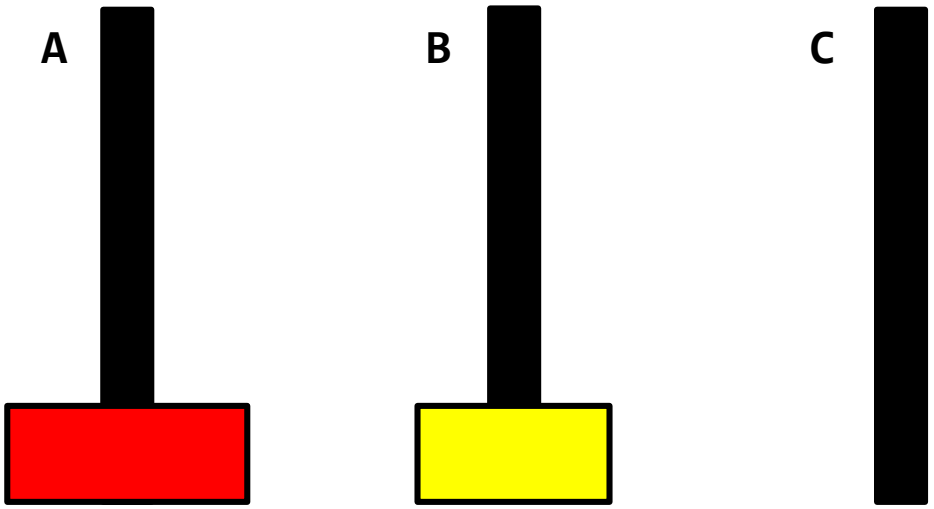
- n = 2



- move disk from A to B

Tower of Hanoi

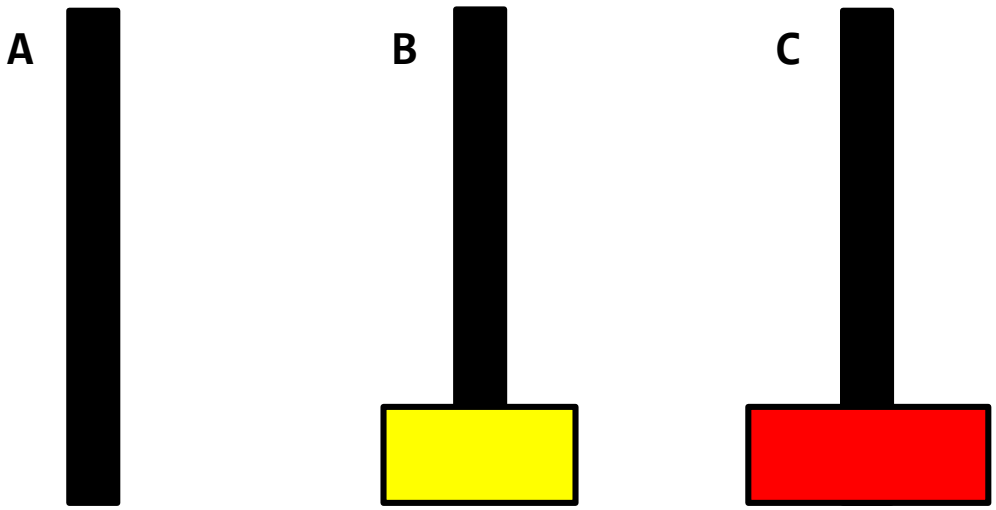
- n = 2



- move disk from A to C

Tower of Hanoi

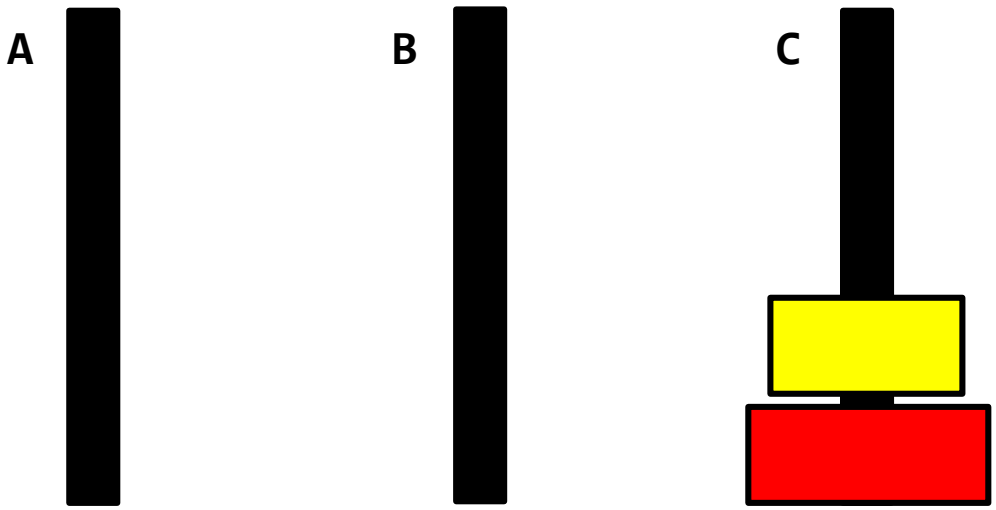
- n = 2



- move disk from B to C

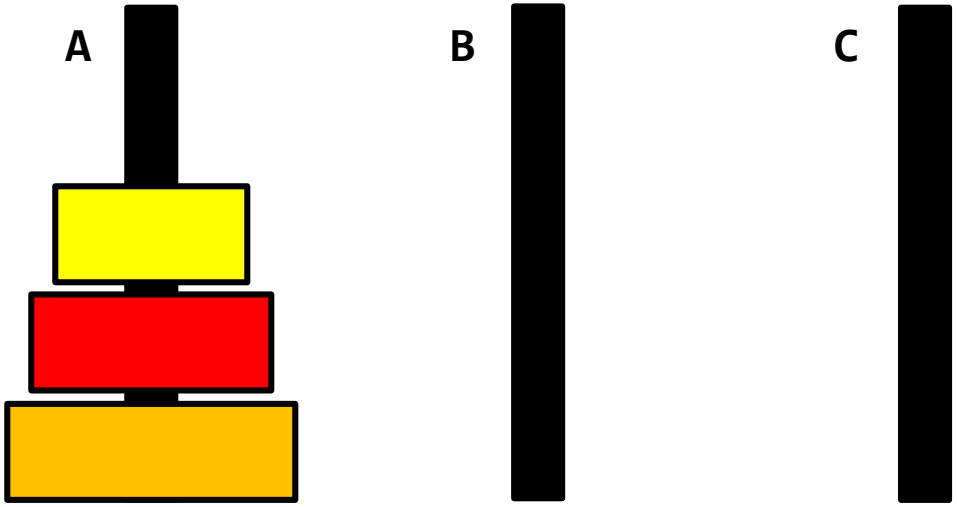
Tower of Hanoi

- $n = 2$



Tower of Hanoi

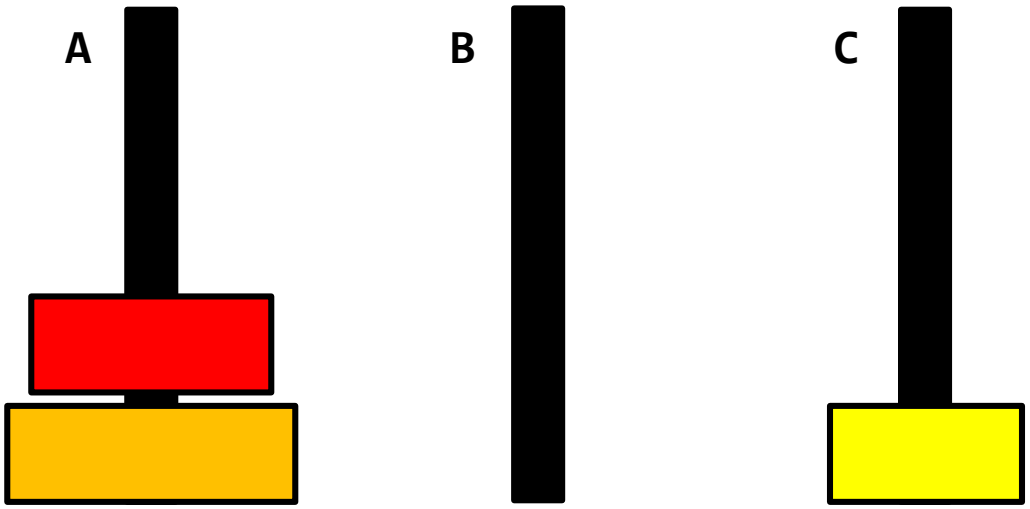
- n = 3



- move disk from A to C

Tower of Hanoi

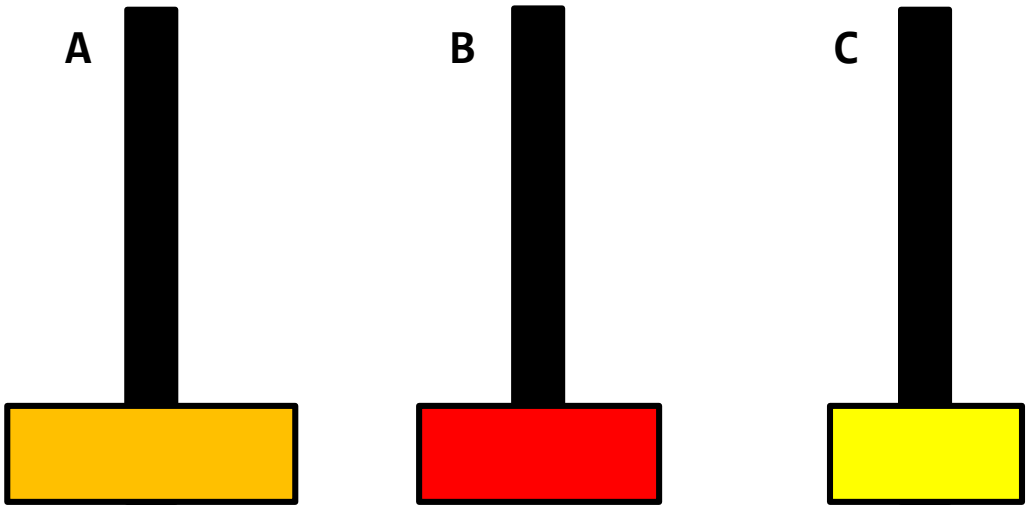
- n = 3



- move disk from A to B

Tower of Hanoi

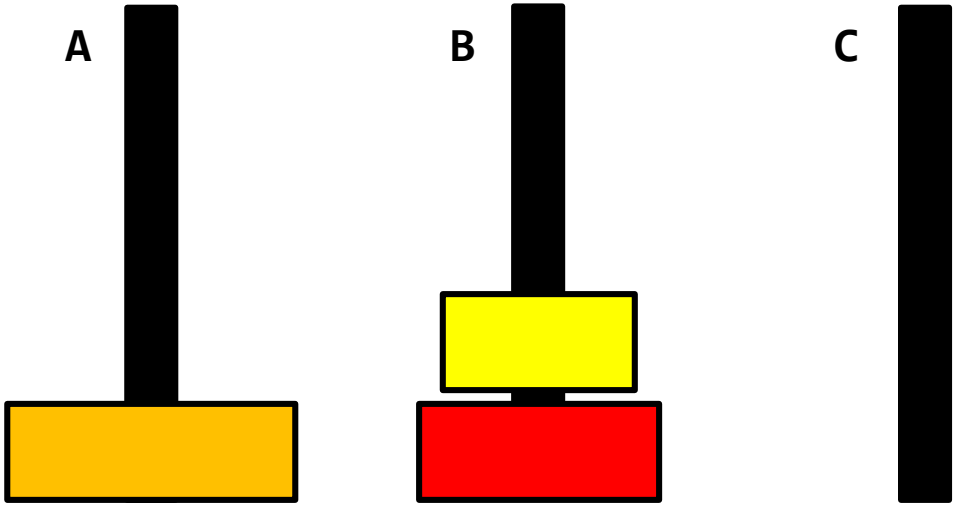
- n = 3



- move disk from C to B

Tower of Hanoi

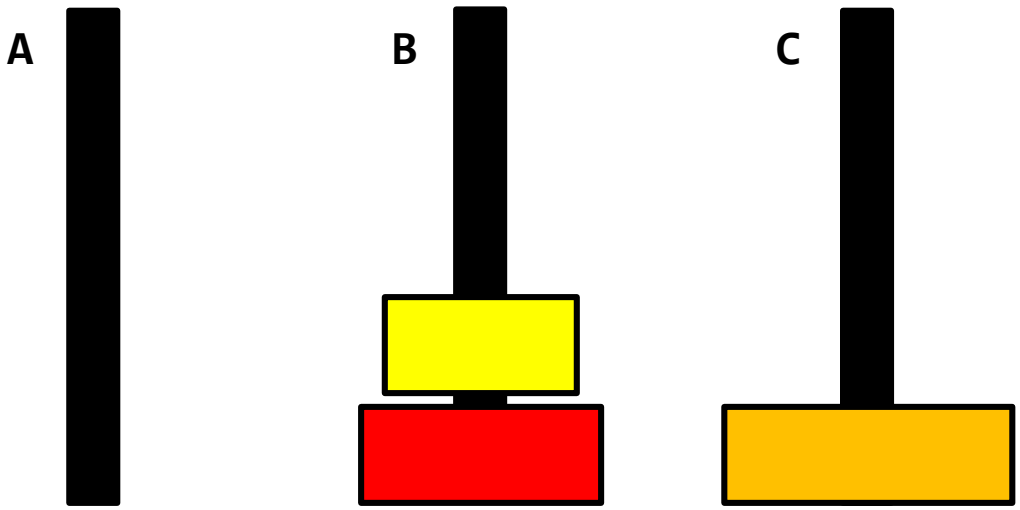
- n = 3



- move disk from A to C

Tower of Hanoi

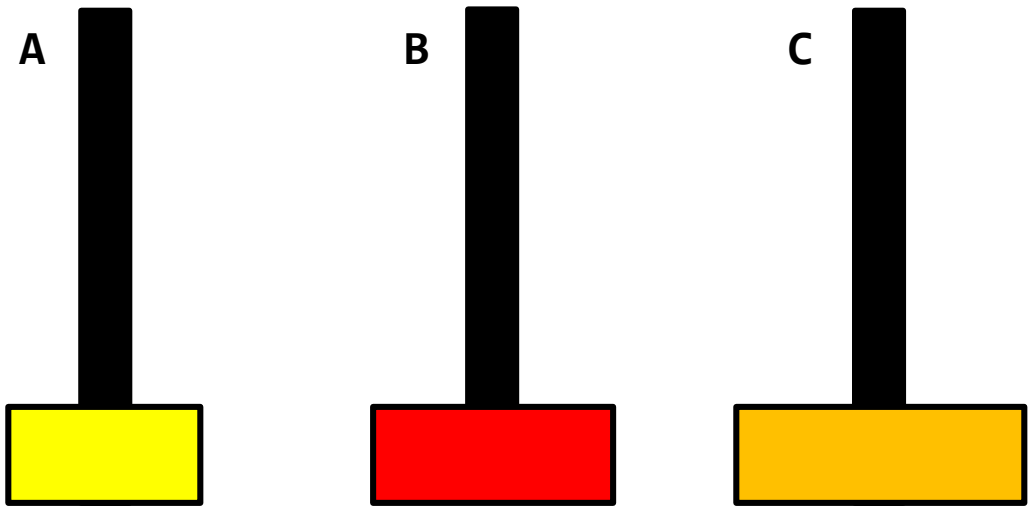
- n = 3



- move disk from B to A

Tower of Hanoi

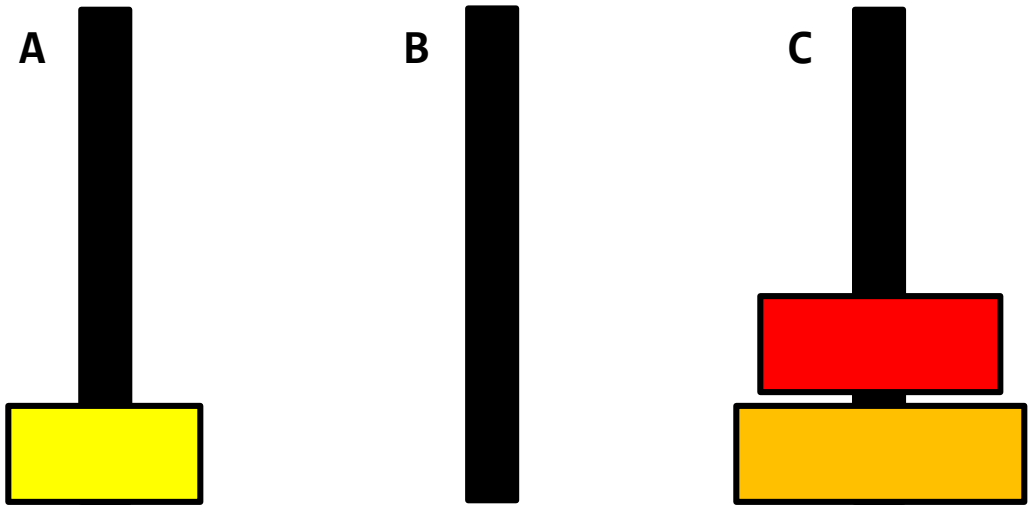
- n = 3



- move disk from B to C

Tower of Hanoi

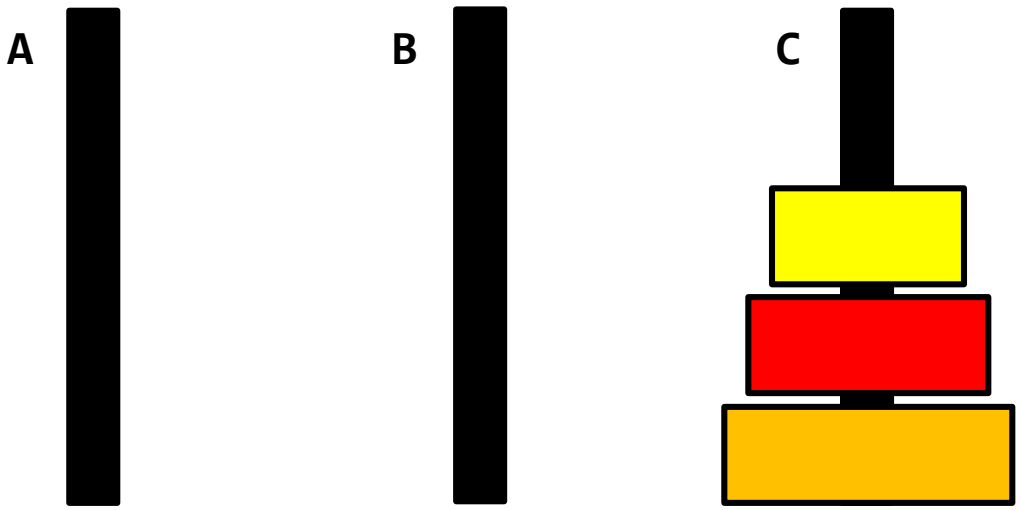
- n = 3



- move disk from A to C

Tower of Hanoi

- n = 3



Tower of Hanoi

- write a loop-based method to solve the Tower of Hanoi problem
  - discuss amongst yourselves now…

Tower of Hanoi

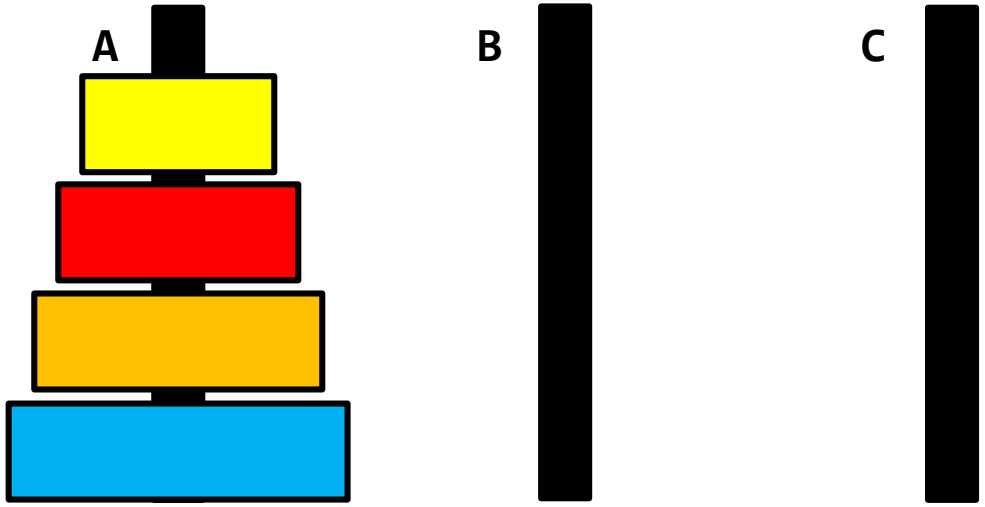
- imagine that you had the following method (see next slide)
  - how would you use the method to solve the Tower of Hanoi problem?
    - discuss amongst yourselves now…

Tower of Hanoi

```
/**
 * Prints the sequence of moves required to move n disks from the
 * starting pole (from) to the goal pole (to) using a third pole
 * (using).
 *
 * @param n
 *          the number of disks to move
 * @param from
 *          the starting pole
 * @param to
 *          the goal pole
 * @param using
 *          a third pole
 * @pre. n is greater than 0
 */
public static void move(int n, String from, String to, String using)
```

Tower of Hanoi

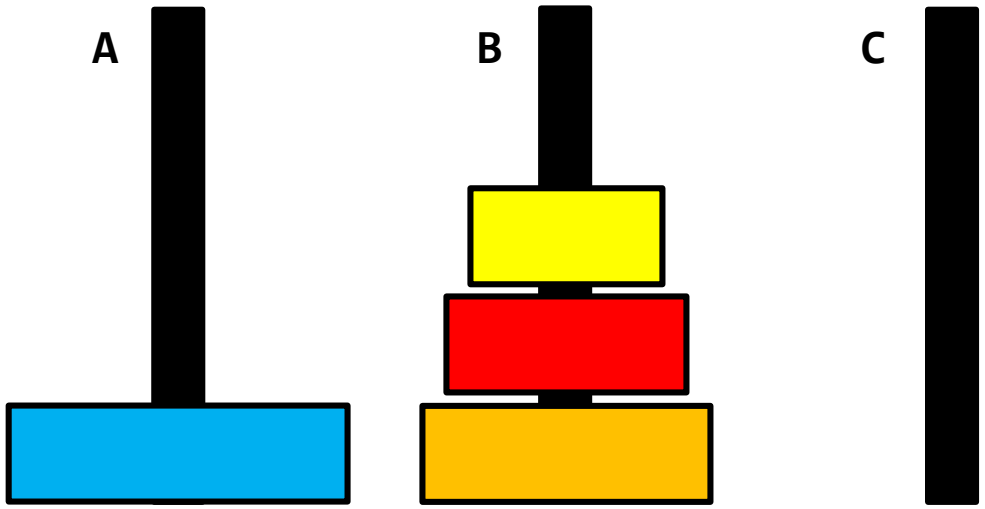
- n = 4



- you eventually end up at (see next slide)…

Tower of Hanoi

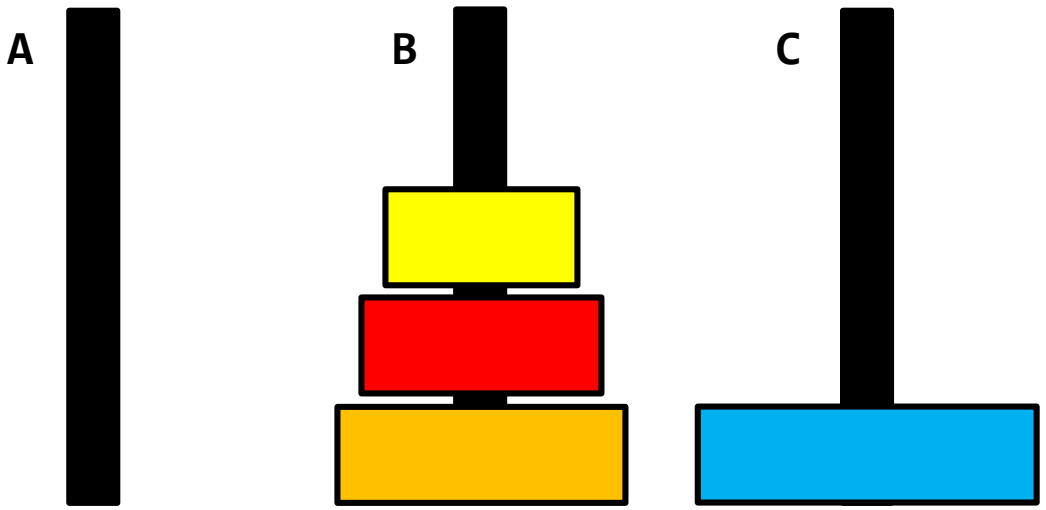
- n = 4



- move disk from A to C

Tower of Hanoi

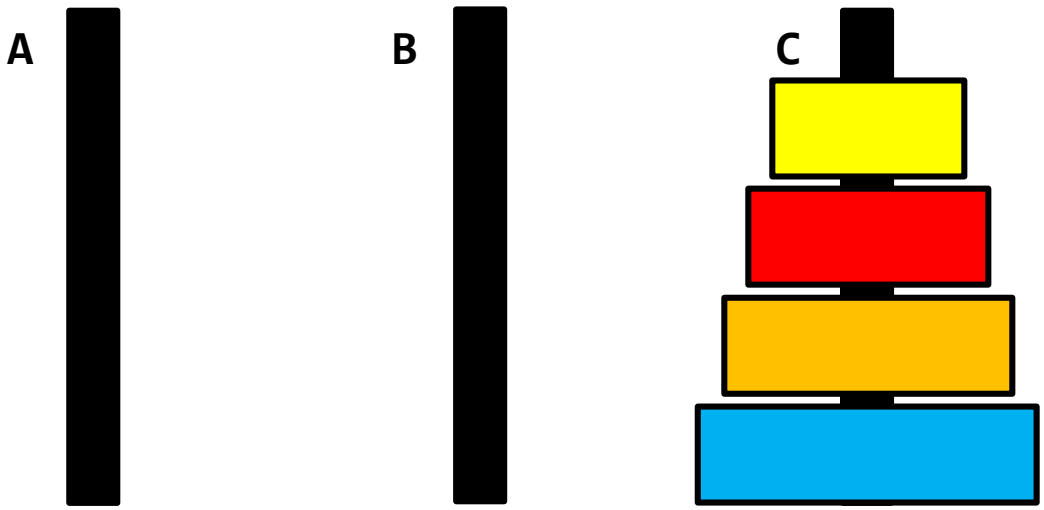
- n = 4



- move (n – 1) disks from B to C using A

Tower of Hanoi

- n = 4



Tower of Hanoi

- notice that to solve the n = 4 size problem, you have to solve a variation of the n = 3 size problem twice and a variation of the n = 1 size problem once
- we can use the **move** method to solve these 3 sub-problems

Tower of Hanoi

- the basic solution can be described as follows:
  1. move (n – 1) disks from A to B
  2. move 1 disk from A to C
  3. move (n – 1) disks from B to C

- furthermore:
  - if exactly n == 1 disk is moved, print out the starting pole and the goal pole for the move

Tower of Hanoi

```
public static void move(int n, String from, String to, String using) {
  if (n == 1) {
    System.out.println("move disk from " + from + " to " + to);
  }
  else {
    move(n - 1, from, using, to);
    move(1, from, to, using);
    move(n - 1, using, to, from);
  }
}
```

Printing n of Something

- suppose you want to implement a method that prints out n copies of a string

```
public static void printIt(String s, int n) {
  for(int i = 0; i < n; i++) {
    System.out.print(s);
  }
}
```

A Different Solution

- alternatively we can use the following algorithm:
  1. if n == 0 done, otherwise
     I. print the string once
     II. print the string (n – 1) more times

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);    // method invokes itself
  }
}
```

Recursion

- a method that calls itself is called a *recursive* method
- a recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printItToo("*", 5)
*printItToo ("*", 4)
**printItToo ("*", 3)
***printItToo ("*", 2)
****printItToo ("*", 1)
*****printItToo ("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Infinite Recursion

- if the base case(s) is missing, or never reached, a recursive method will run forever (or until the computer runs out of resources)

```
public static void printItForever(String s, int n) {
  // missing base case; infinite recursion
  System.out.print(s);
  printItForever(s, n - 1);
}
```

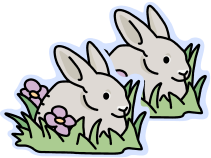
```
printItForever("*", 1)
* printItForever("*", 0)
** printItForever("*", -1)
*** printItForever("*", -2) ..........
```

Climbing a Flight of n Stairs

- not Java

```
/**
  * method to climb n stairs
  */
climb(n) :
if n == 0
  done
else
  step up 1 stair
  climb(n – 1);
end
```

Rabbits

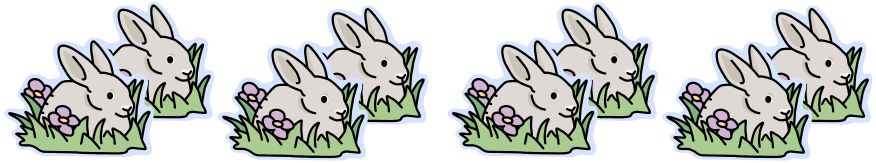


Month 0: 1 pair

0 additional pairs

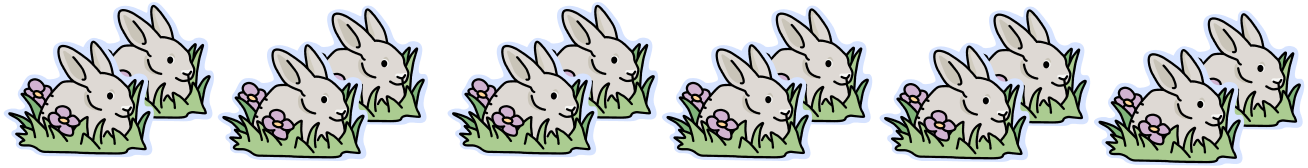
Month 1: first pair makes another pair

1 additional pair



Month 2: each pair makes another pair; oldest pair dies

1 additional pair



Month 3: each pair makes another pair; oldest pair dies

2 additional pairs

Fibonacci Numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n – 1) + F(n – 2)`

Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

Recursive Methods & Return Values

- write a recursive method that multiplies two positive integer values (i.e., both values are strictly greater than zero)

- observation: $m \times n$ means add m n's together
  - in other words, you can view multiplication as recursive addition

Recursive Methods & Return Values

- not Java:

```
/**
 * Computes m * n
 */
multiply(m, n) :
if m == 1
    return n
else
    return n + multiply(m - 1, n)
```

```
public static int multiply(int m, int n) {
  if (m == 1) {
    return n;
  }
  return n + multiply(m - 1, n);
}
```

Recursive Methods & Return Values

- example: write a recursive method **`countZeros`** that counts the number of zeros in an integer number **`n`**
  - **`10305060700002L`** has 8 zeros

- trick: examine the following sequence of numbers
  1. **`10305060700002`**
  2. **`1030506070000`**
  3. **`103050607000`**
  4. **`10305060700`**
  5. **`103050607`**
  6. **`1030506 ...`**

Recursive Methods & Return Values

- not Java:

```
/**
 * Counts the number of zeros in an integer n
 */
countZeros(n) :
if the last digit in n is a zero
    return 1 + countZeros(n / 10)
else
    return countZeros(n / 10)
```

- don't forget to establish the base case(s)
  - when should the recursion stop? when you reach a single digit (not zero digits; you never reach zero digits!)
    - base case #1 : `n == 0`
      - `return 1`
    - base case #2 : `n != 0 && n < 10`
      - `return 0`

```
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

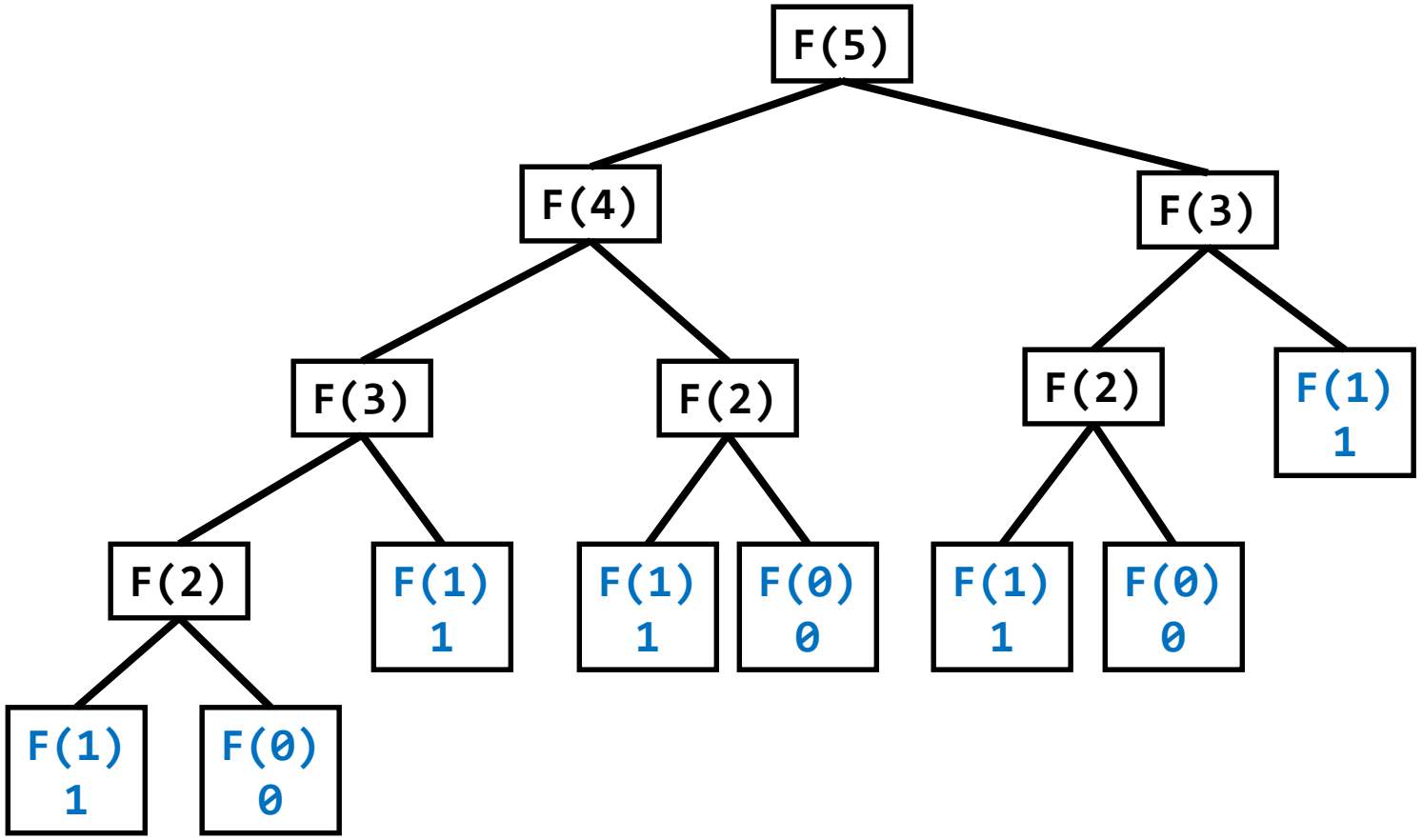
countZeros Call Stack

`callZeros( 800410L )`

last in first out

`callZeros( 8L )`	`0`
`callZeros( 80L )`	`1 + 0`
`callZeros( 800L )`	`1 + 1 + 0`
`callZeros( 8004L )`	`0 + 1 + 1 + 0`
`callZeros( 80041L )`	`0 + 0 + 1 + 1 + 0`
`callZeros( 800410L )`	`1 + 0 + 0 + 1 + 1 + 0`
	`= 3`

Fibonacci Call Tree



Compute Powers of 10

- write a recursive method that computes 10^n for any integer value **n**
- recall:
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$
  - $10^{-n} = 1 / 10^n$

```
public static double powerOf10(int n) {
  if (n == 0) {
    // base case
    return 1.0;
  }
  else if (n > 0) {
    // recursive call for positive n
    return 10.0 * powerOf10(n - 1);
  }
  else {
    // recursive call for negative n
    return 1.0 / powerOf10(-n);
  }
}
```

Fibonacci Numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

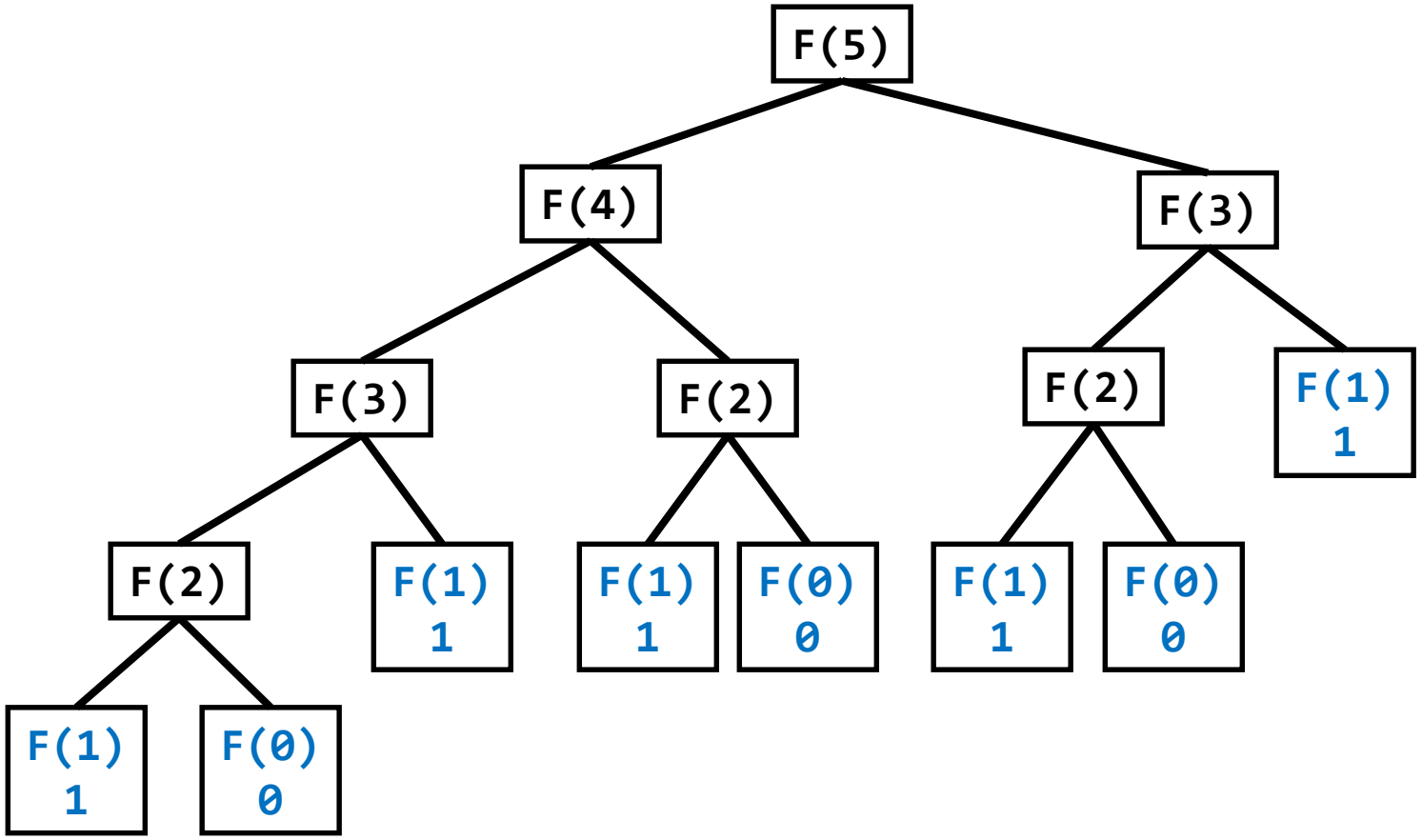
- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n – 1) + F(n – 2)`

Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

Fibonacci Call Tree

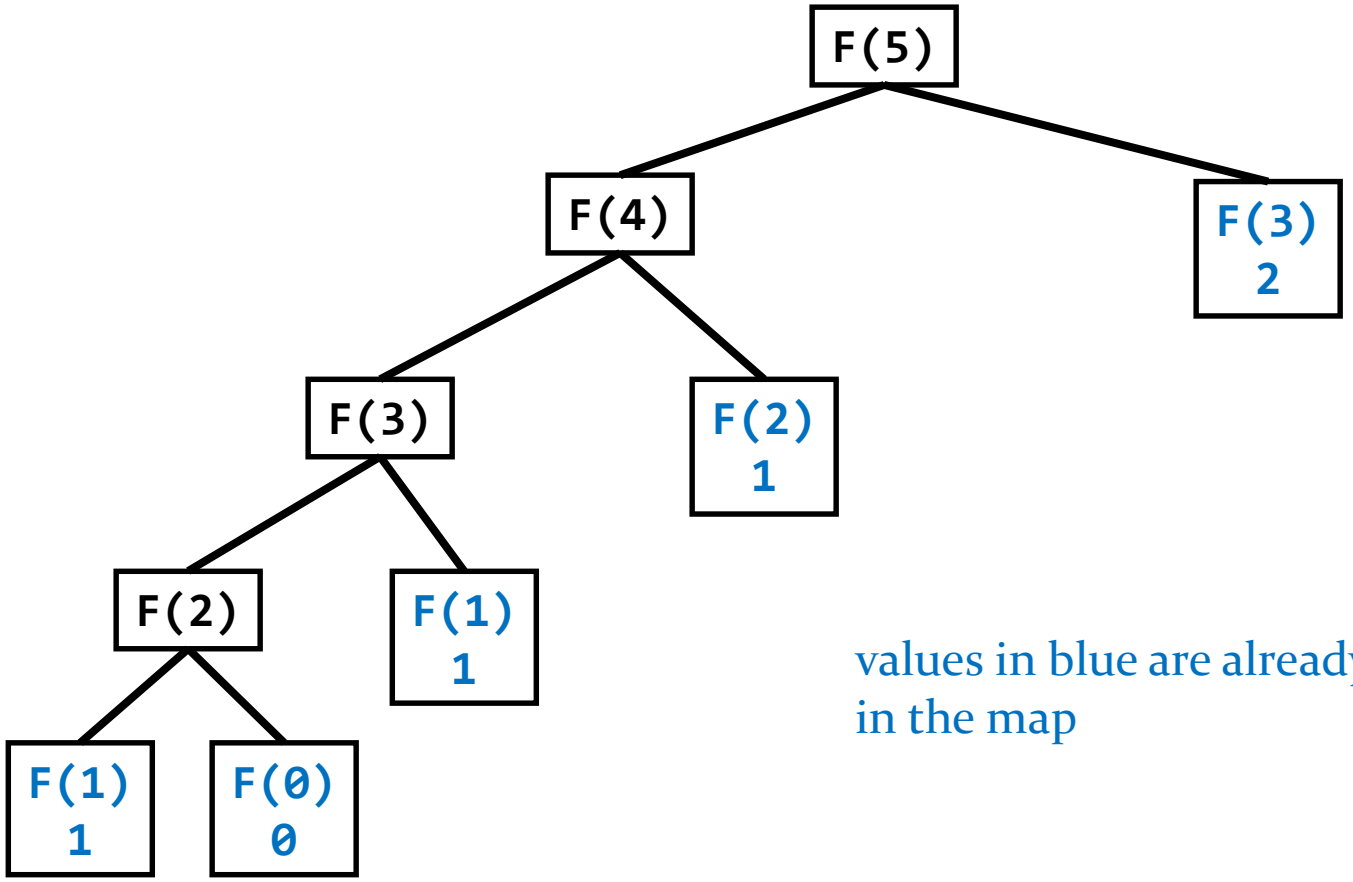


A Better Recursive Fibonacci

```
public class Fibonacci {
  private static Map<Integer, Long> values = new HashMap<Integer, Long>();
  static {
    Fibonacci.values.put(0, (long) 0);
    Fibonacci.values.put(1, (long) 1);
  }

  public static long getValue(int n) {
    Long value = Fibonacci.values.get(n);
    if (value != null) {
      return value;
    }
    value = Fibonacci.getValue(n - 1) + Fibonacci.getValue(n - 2);
    Fibonacci.values.put(n, value);
    return value;
  }
}
```

Better Fibonacci Call Tree



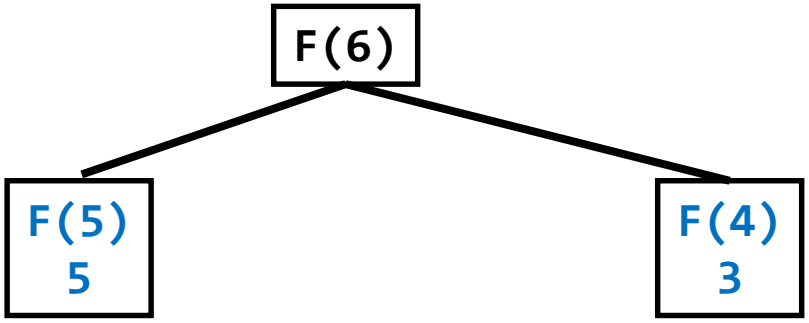
values in blue are already stored in the map

A Better Recursive Fibonacci

- because the map is static subsequent calls to **Fibonacci.*getValue*(int)** can use the values already computed and stored in the map

Better Fibonacci Call Tree

- assuming the client has already invoked **Fibonacci.*getValue*(5)**



values in blue are already stored in the map

Compute Powers of 10

- write a recursive method that computes 10^n for any integer value **n**
- recall:
  - $10^n = 1 / 10^{-n}$ if **n < 0**
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$

```
public static double powerOf10(int n) {
  if (n < 0) {
    return 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    return 1.0;
  }
  return n * powerOf10(n - 1);
}
```

A Better Powers of 10

- recall:
  - $10^n = 1 / 10^{-n}$ if $n < 0$
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$ if **n** is odd
  - $10^n = 10^{n/2} * 10^{n/2}$ if **n** is even

```
public static double powerOf10(int n) {
  if (n < 0) {
    return 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    return 1.0;
  }
  else if (n % 2 == 1) {
    return 10 * powerOf10(n - 1);
  }
  double value = powerOf10(n / 2);
  return value * value;
}
```

What happens during recursion

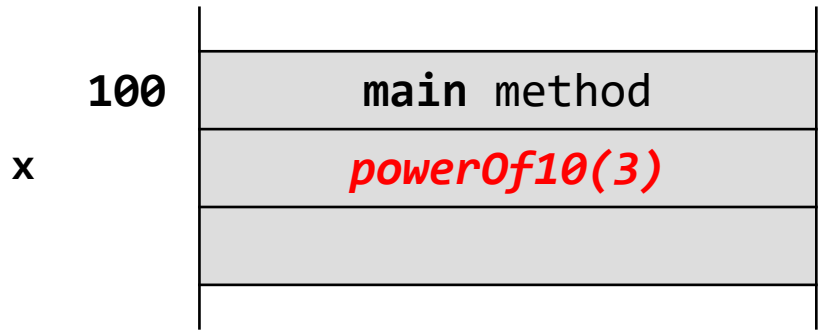
What Happens During Recursion

- a simplified model of what happens during a recursive method invocation is the following:
  - whenever a method is invoked that method runs in a *new* block of memory
    - when a method recursively invokes itself, a new block of memory is allocated for the newly invoked method to run in

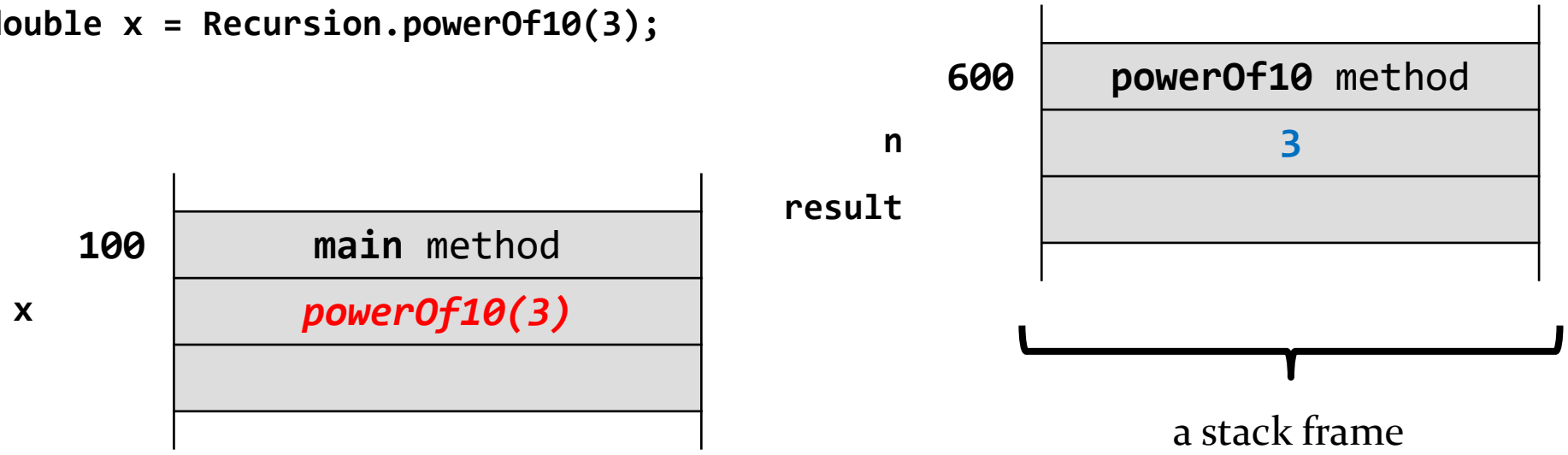
- consider a slightly modified version of the **`powerOf10`** method

```
public static double powerOf10(int n) {
  double result;
  if (n < 0) {
    result = 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    result = 1.0;
  }
  else {
    result = 10 * powerOf10(n - 1);
  }
  return result;
}
```

```
double x = Recursion.powerOf10(3);
```

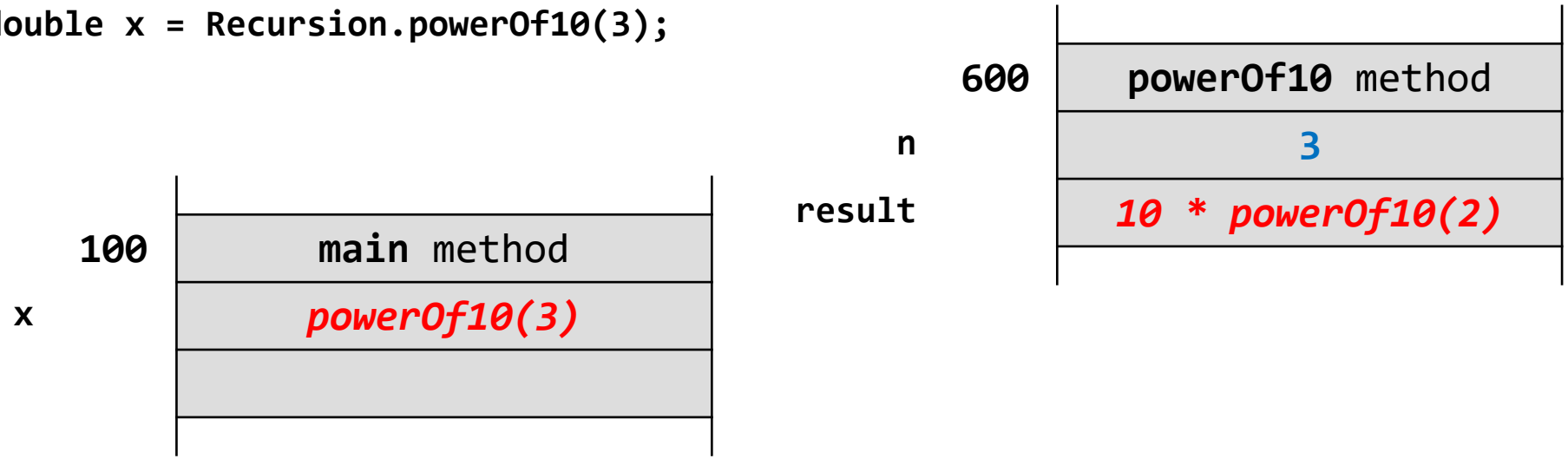


```
double x = Recursion.powerOf10(3);
```

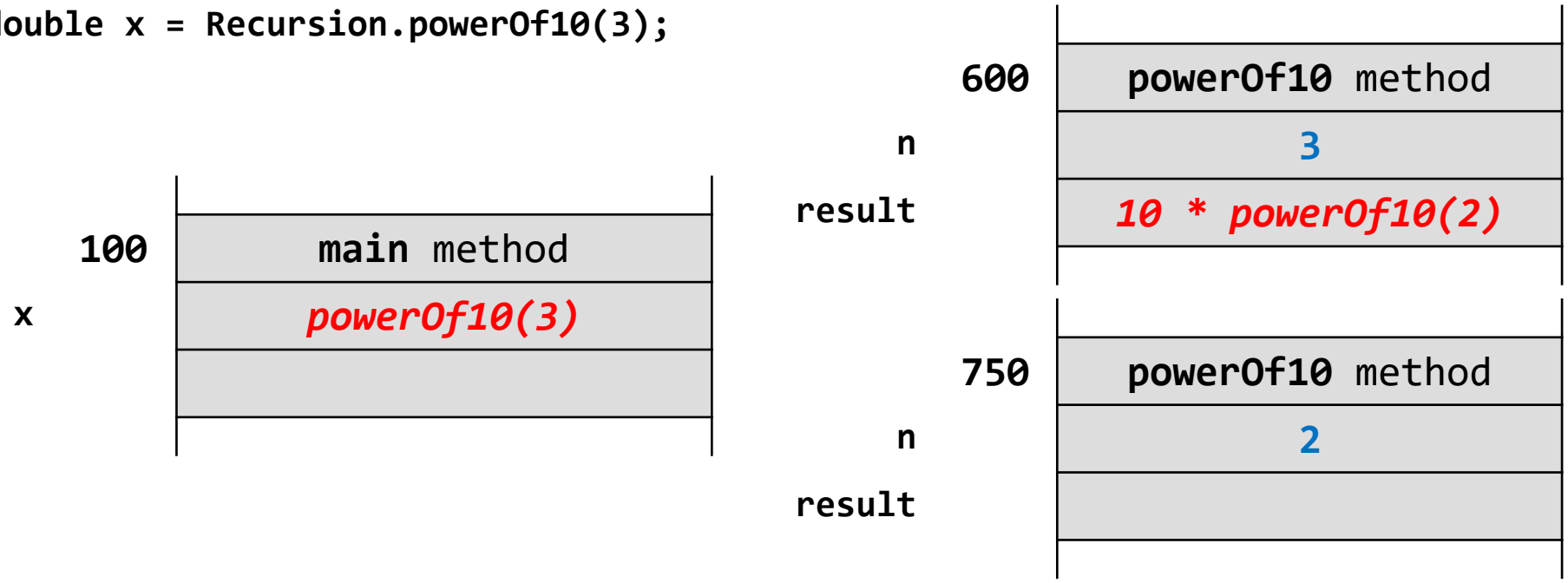


- methods occupy space in a region of memory called the *call stack*
- information regarding the state of the method is stored in a *stack frame*
- the stack frame includes information such as the method parameters, local variables of the method, where the return value of the method should be copied to, where control should flow to after the method completes, ...
- stack memory can be allocated and deallocated very quickly, but this speed is obtained by restricting the total amount of stack memory
- if you try to solve a large problem using recursion you can exceed the available amount of stack memory which causes your program to crash

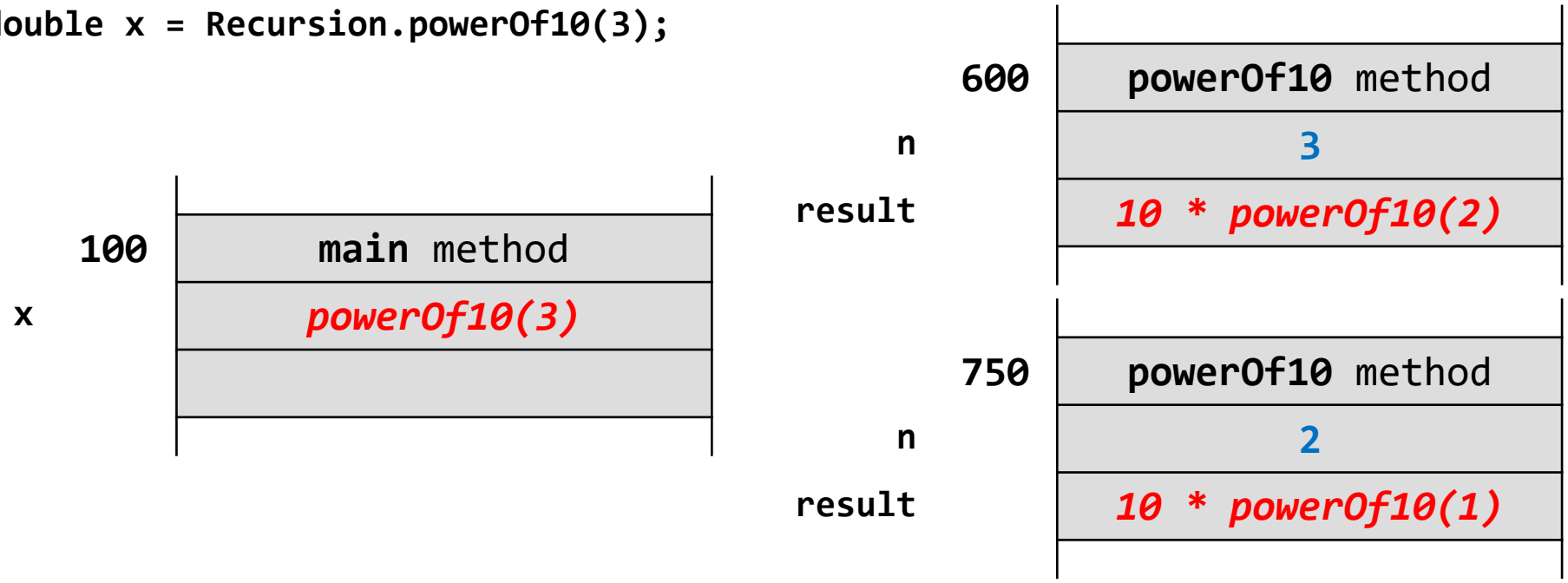
```
double x = Recursion.powerOf10(3);
```



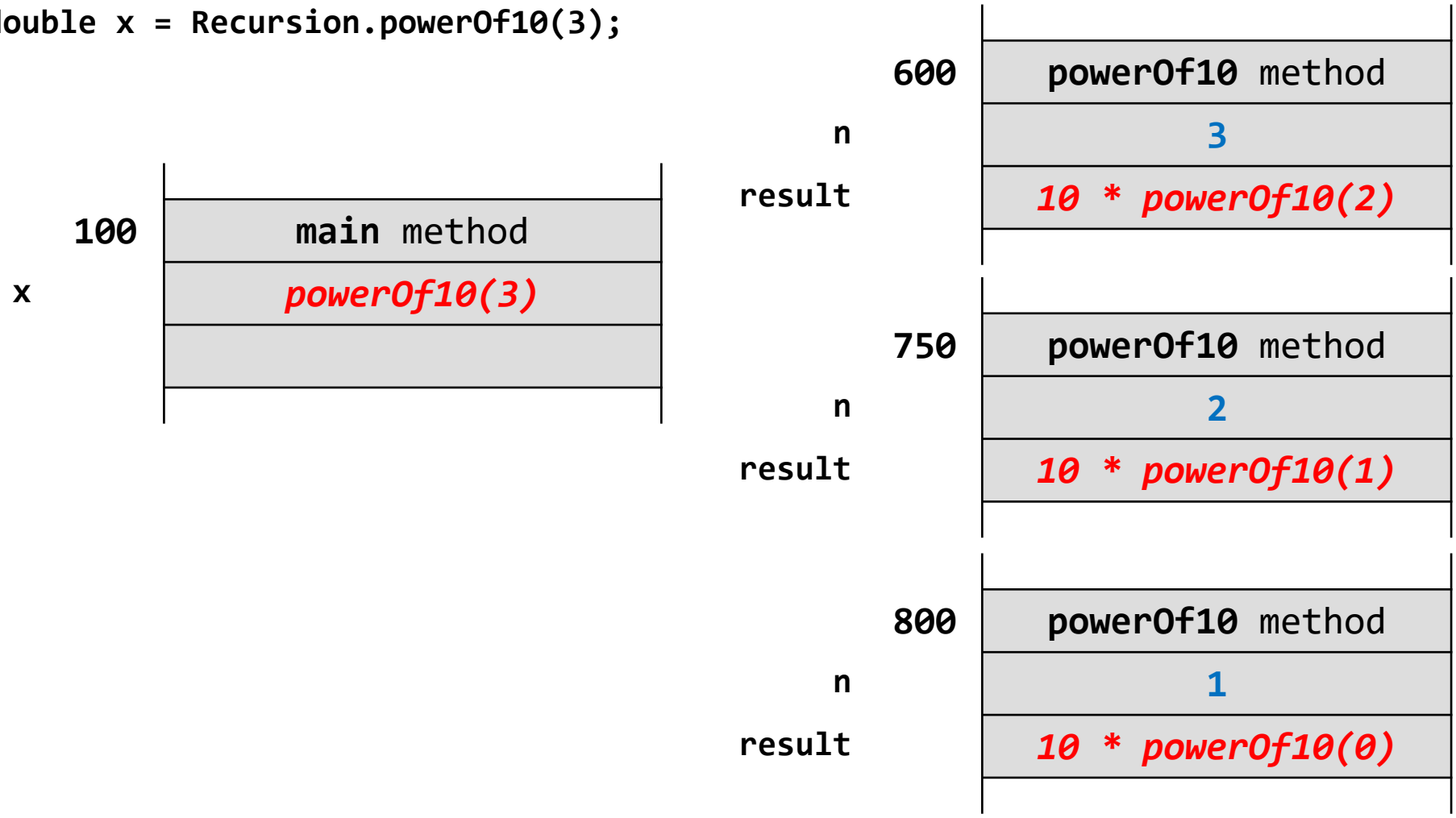
```
double x = Recursion.powerOf10(3);
```



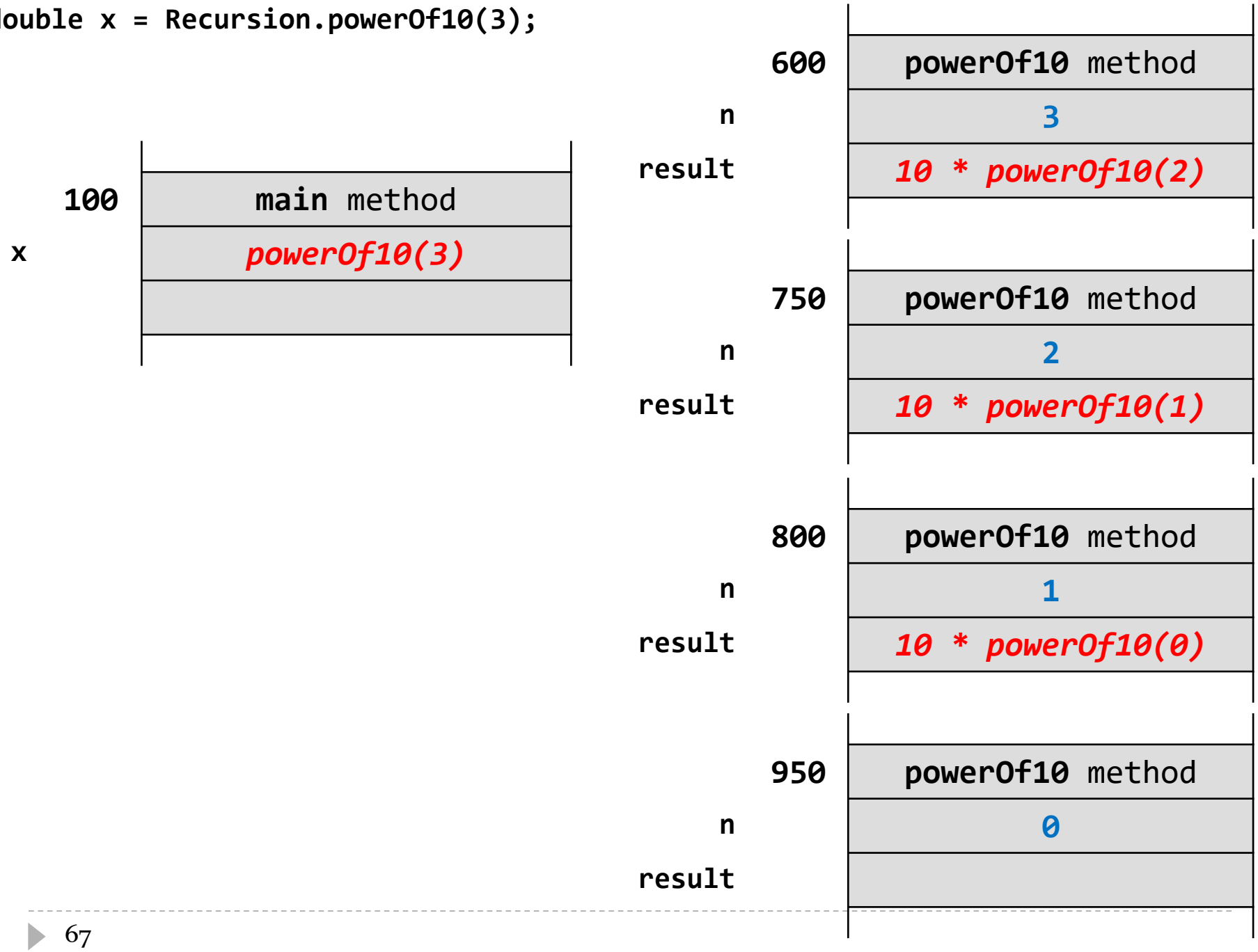
```
double x = Recursion.powerOf10(3);
```



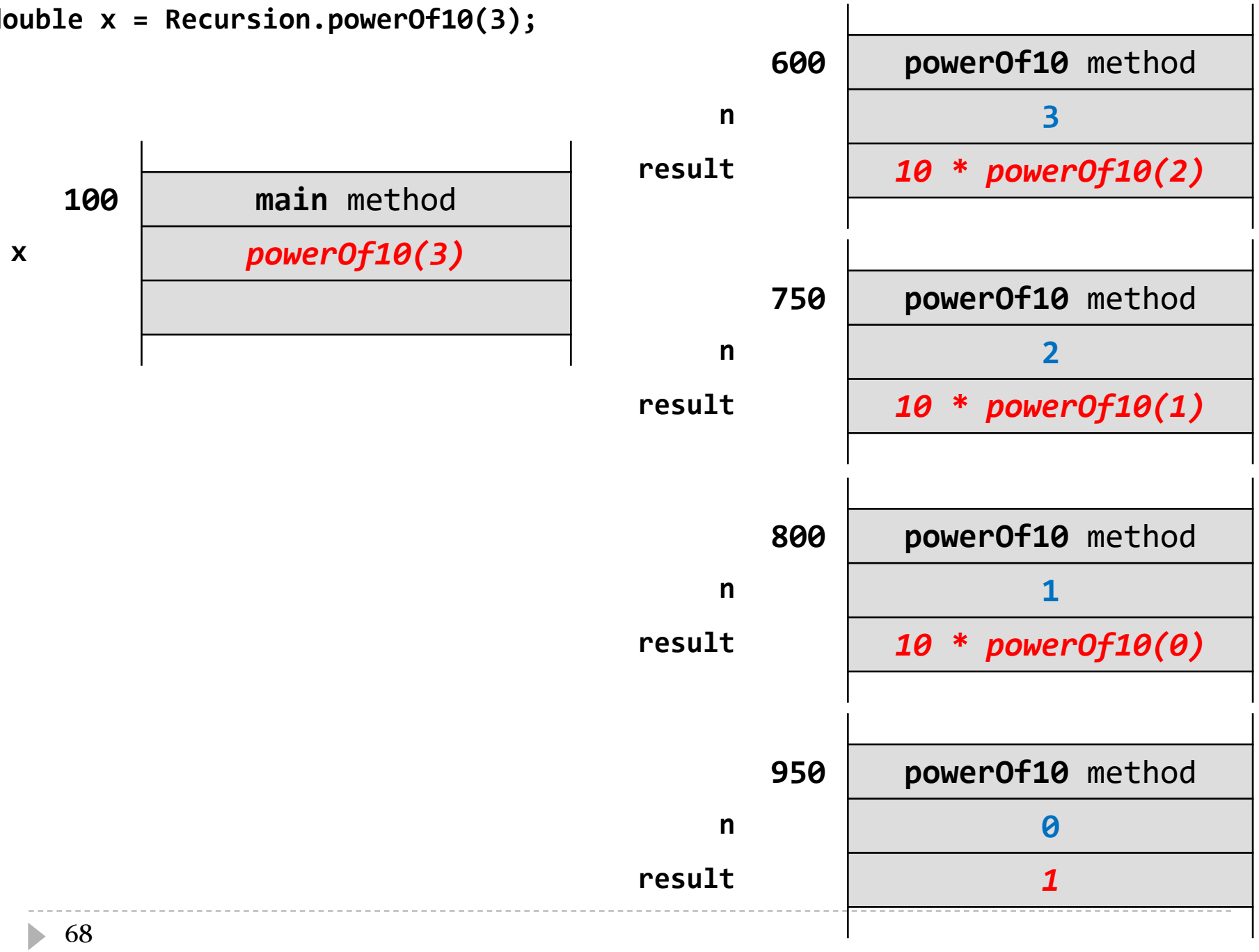
```
double x = Recursion.powerOf10(3);
```



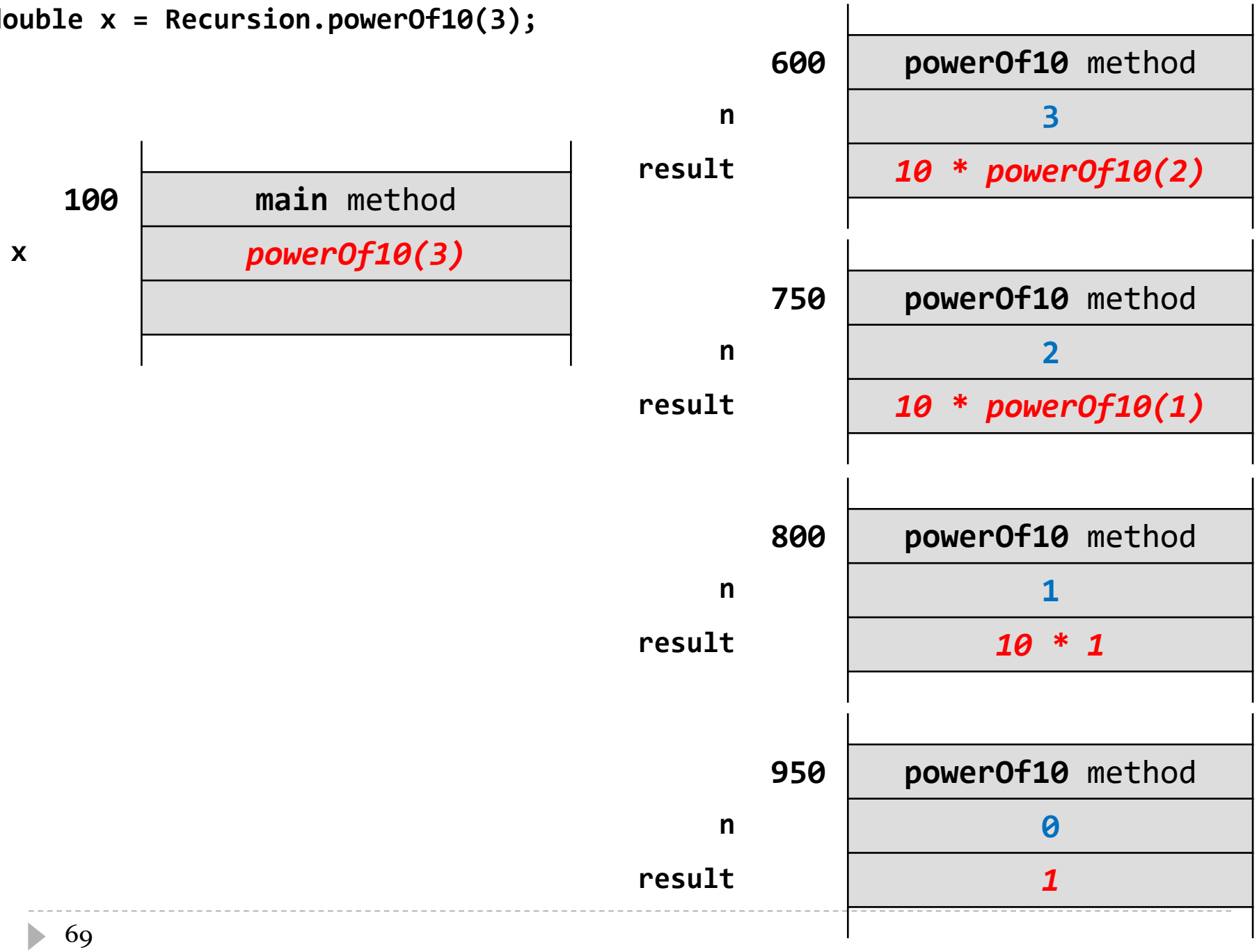
```
double x = Recursion.powerOf10(3);
```



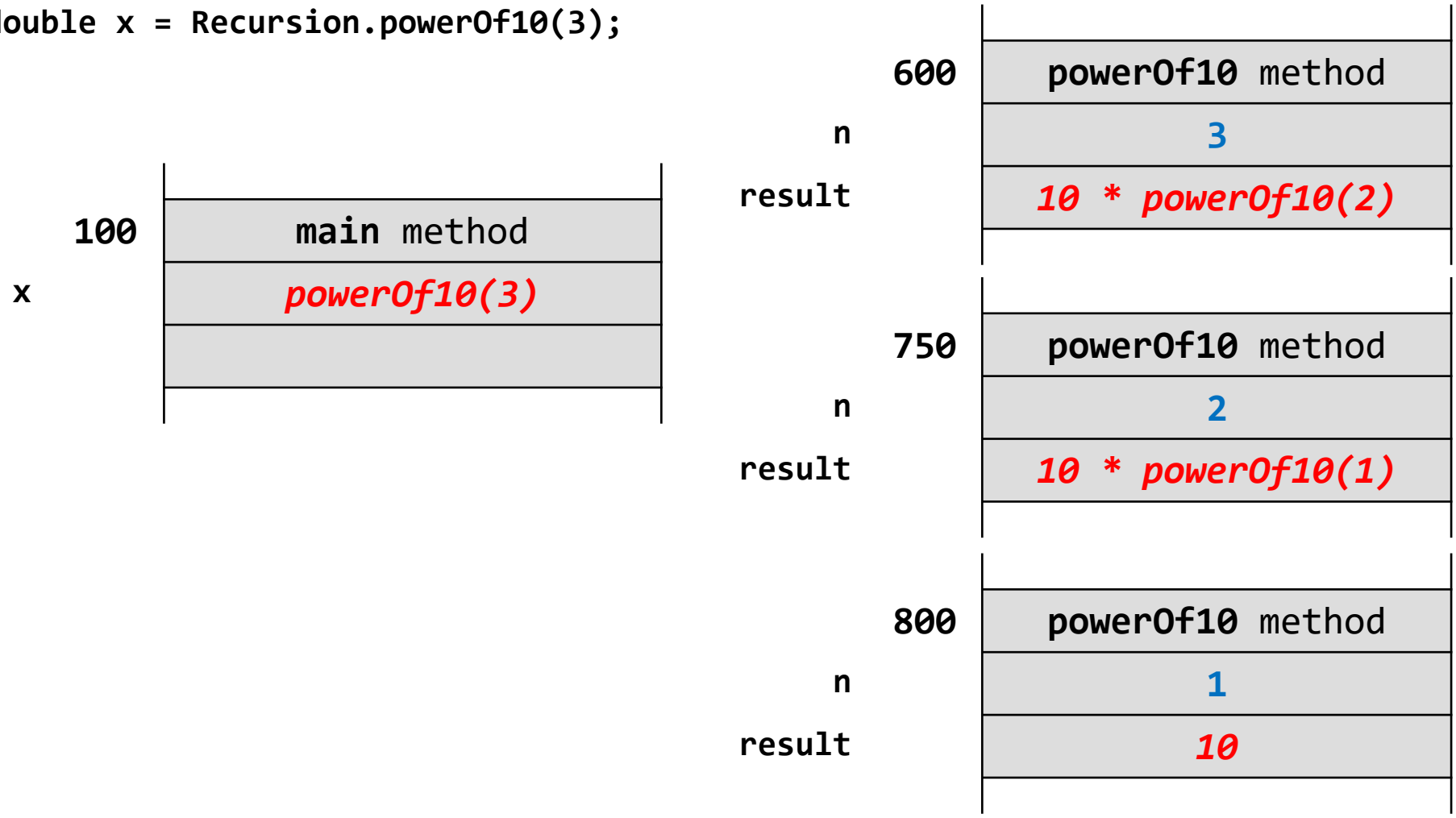
```
double x = Recursion.powerOf10(3);
```



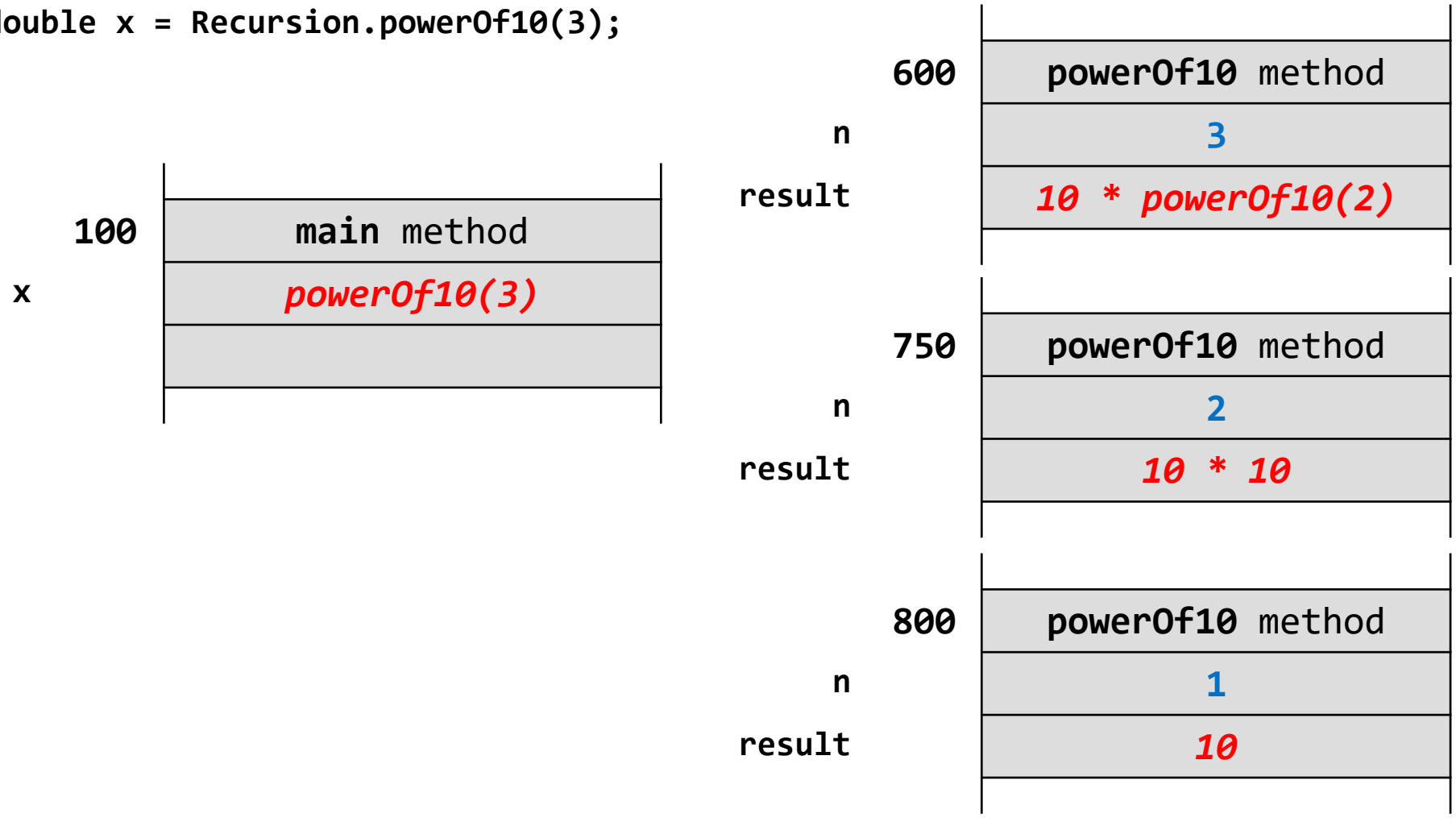
```
double x = Recursion.powerOf10(3);
```



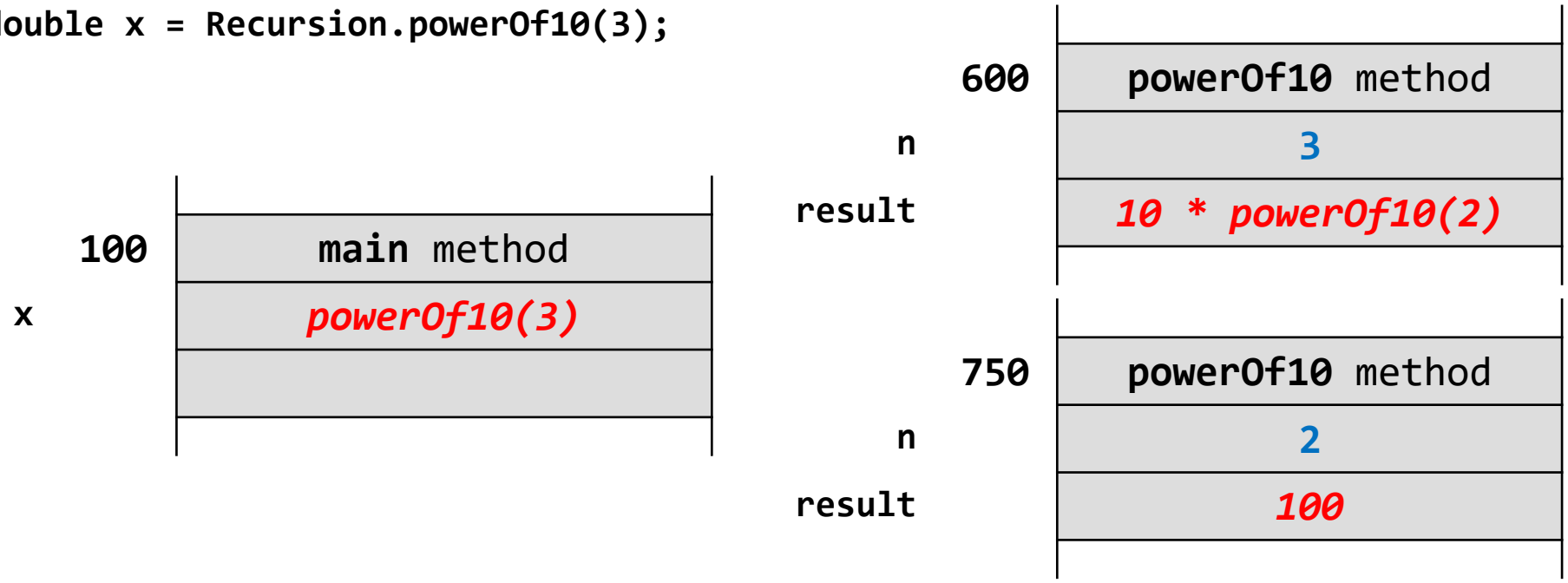
```
double x = Recursion.powerOf10(3);
```



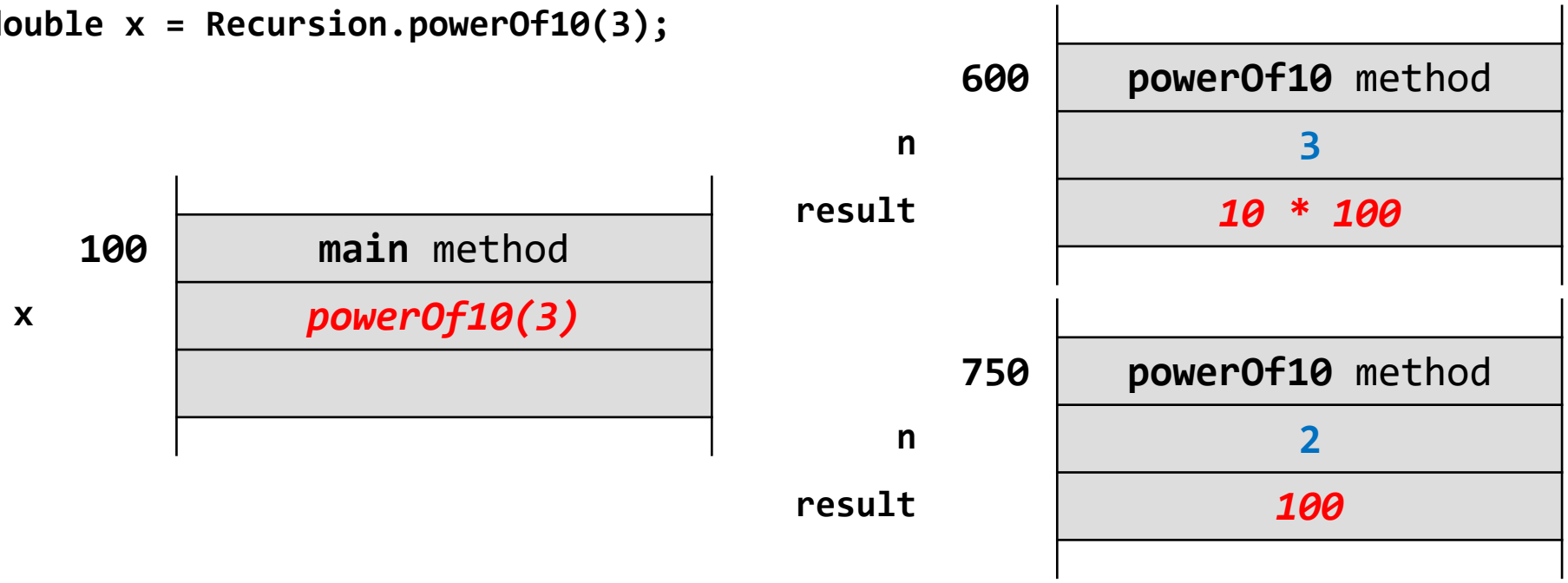
```
double x = Recursion.powerOf10(3);
```



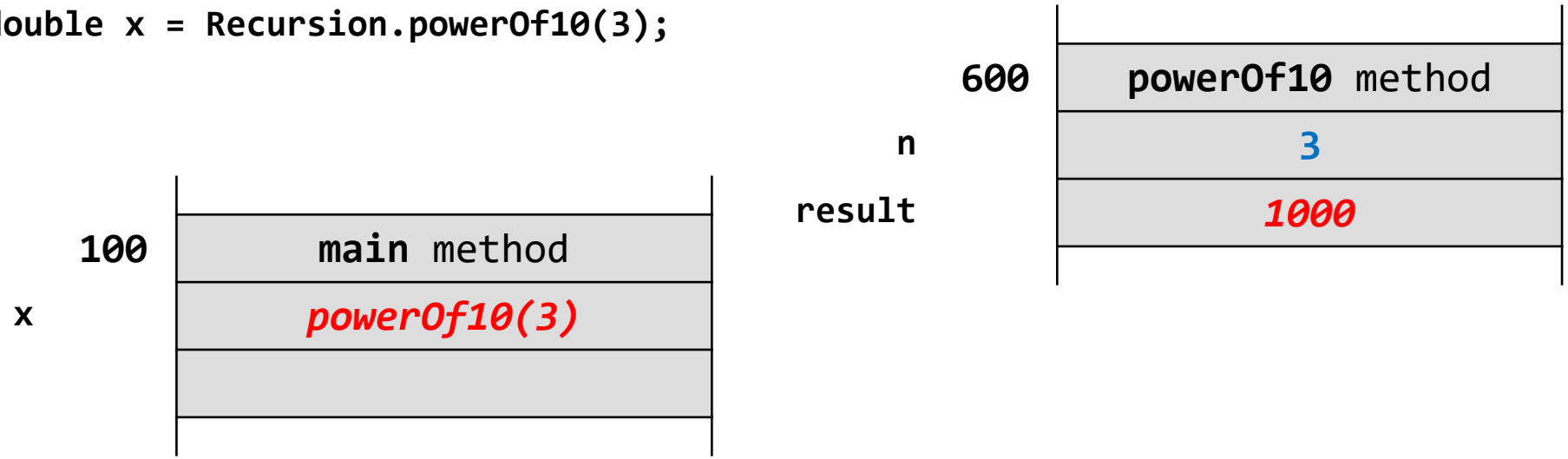
```
double x = Recursion.powerOf10(3);
```



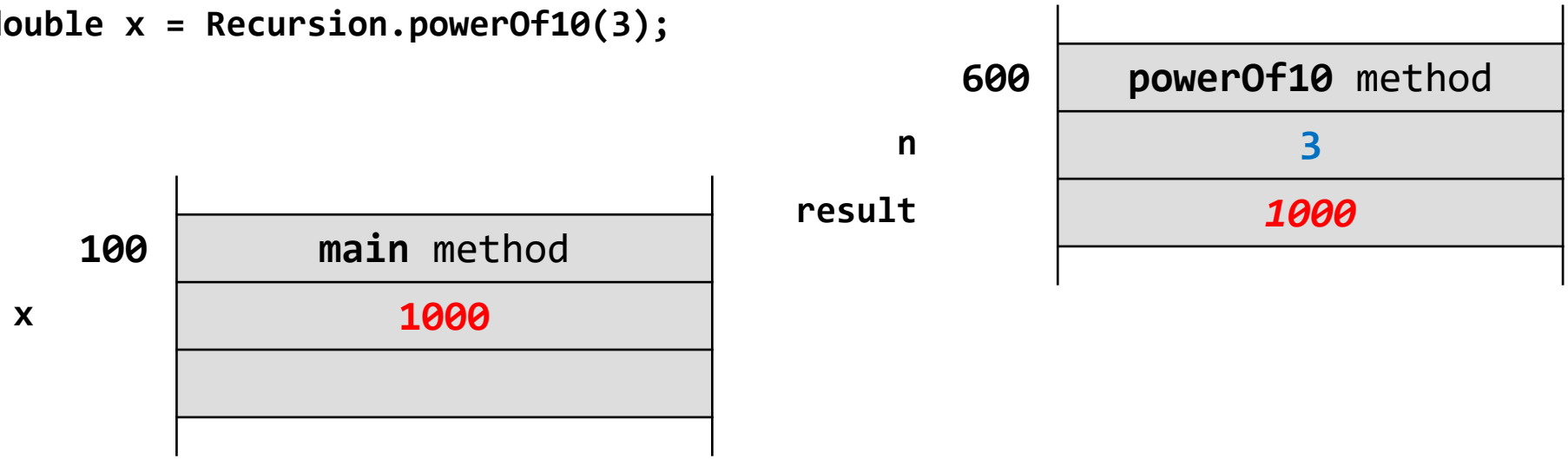
```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```