Some remarks

- In the final, if you do not mark your section number or put the answer to the multiple choice questions in the front page, we'll deduct marks.

- When we ask for a method that expects some input and produces some, it meas:
  - Write a method, not a class
  - Write a method that has as parameters the expected input and returns the output,
  - ... not a method that uses Keyboard to ask the user for input and System.out.println(...) for output.

Input and output

```
boolean isPrime(int n)
{
  boolean result = true;
  int i = 2;
  while (i < n) {
    if (n % i == 0) result = false;
    i++;
  }
  return result;
}
```

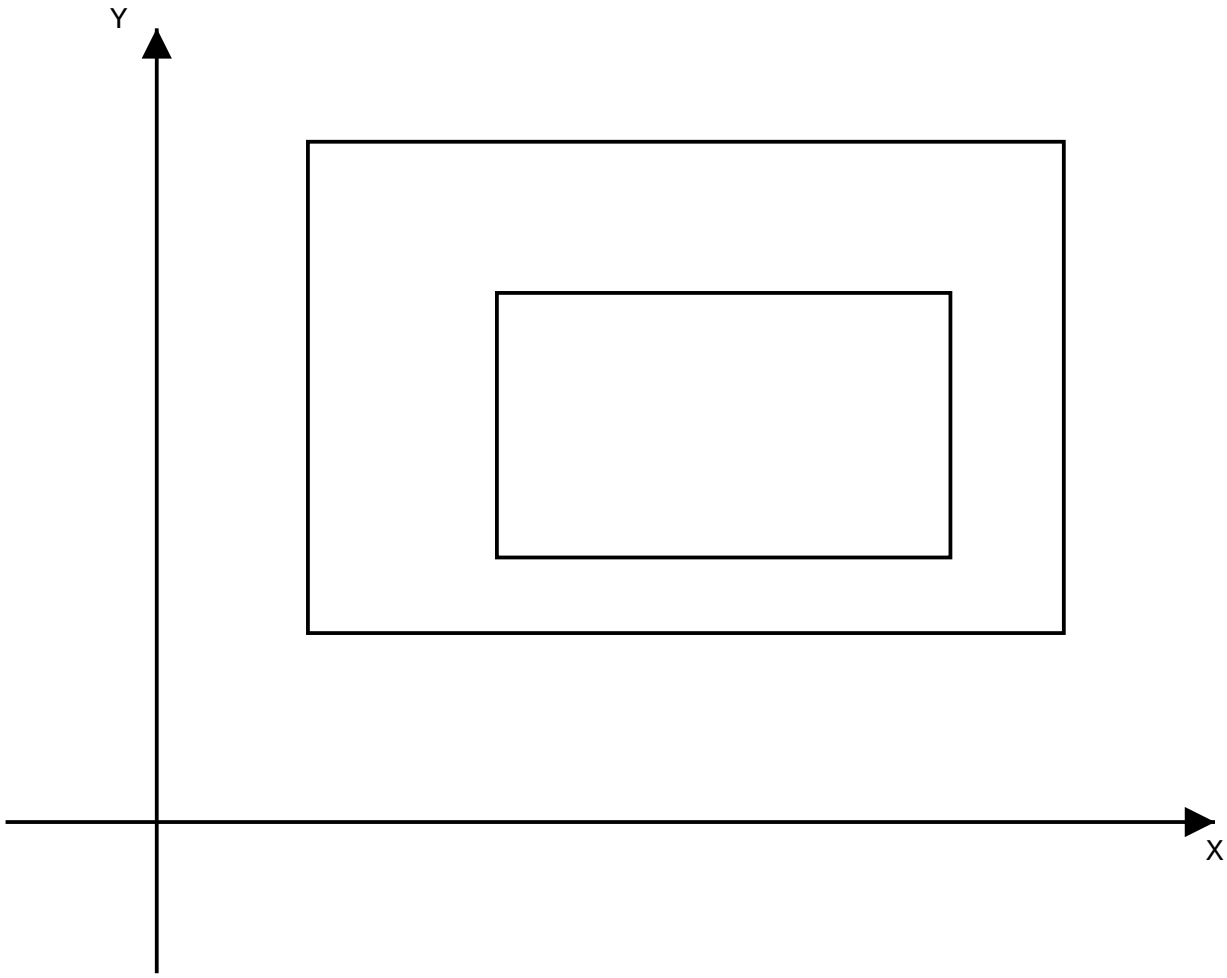
Input and output

```
void isPrime()
{
  int n;
  System.out.print(“Enter a number: ”);
  n = Keyboard.readInt();
  boolean result = true;
  int i = 2;
  while (i < n) {
    if (n % i == 0) result = false;
    i++;
  }
  System.out.println(result);
}
```

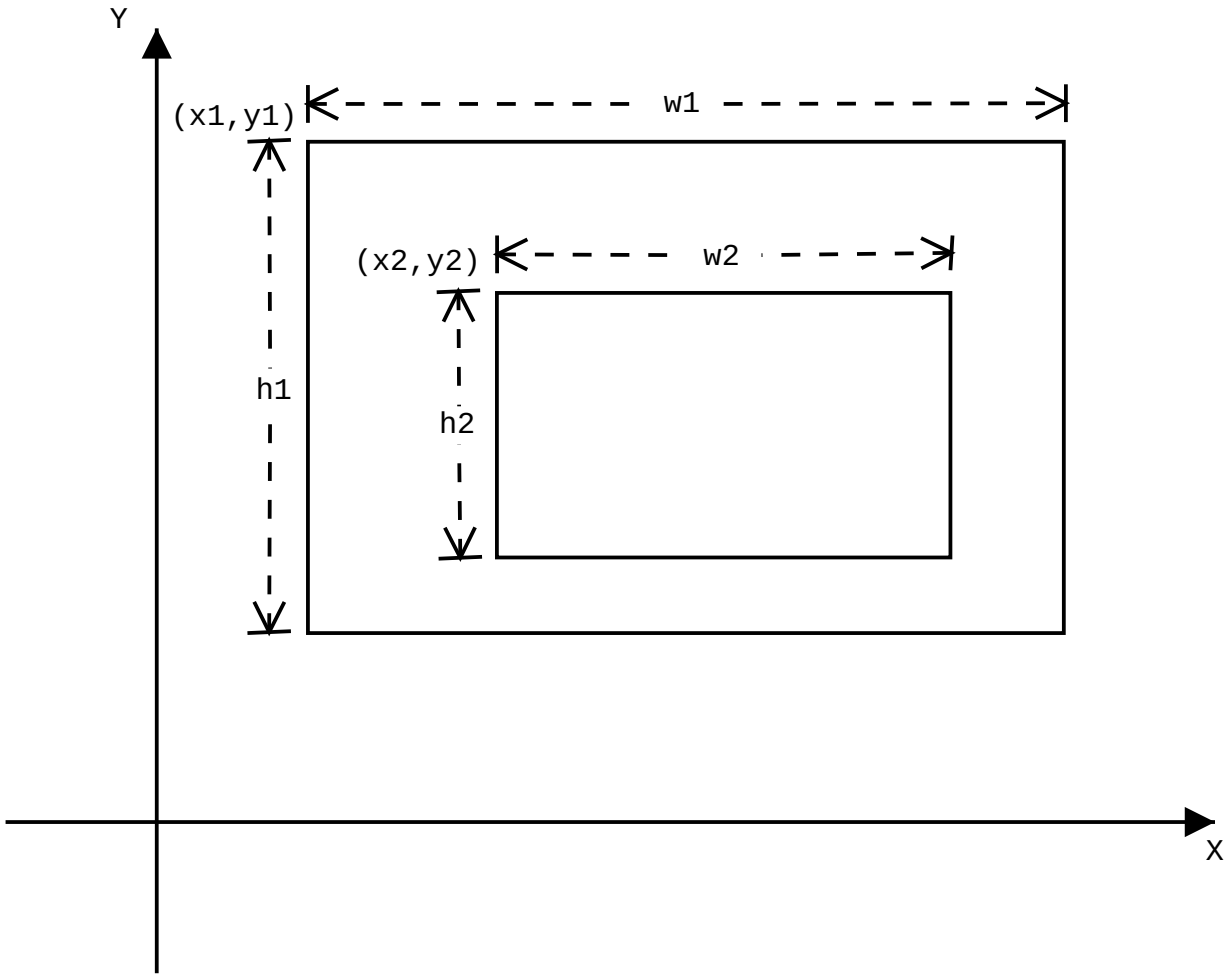
Input and output

```
void print_primes_smaller_than(int m)
{
  int i = 0;
  while (i < m) {
    if (isPrime(i)) {
      System.out.println(i);
    }
    i++;
  }
}
```

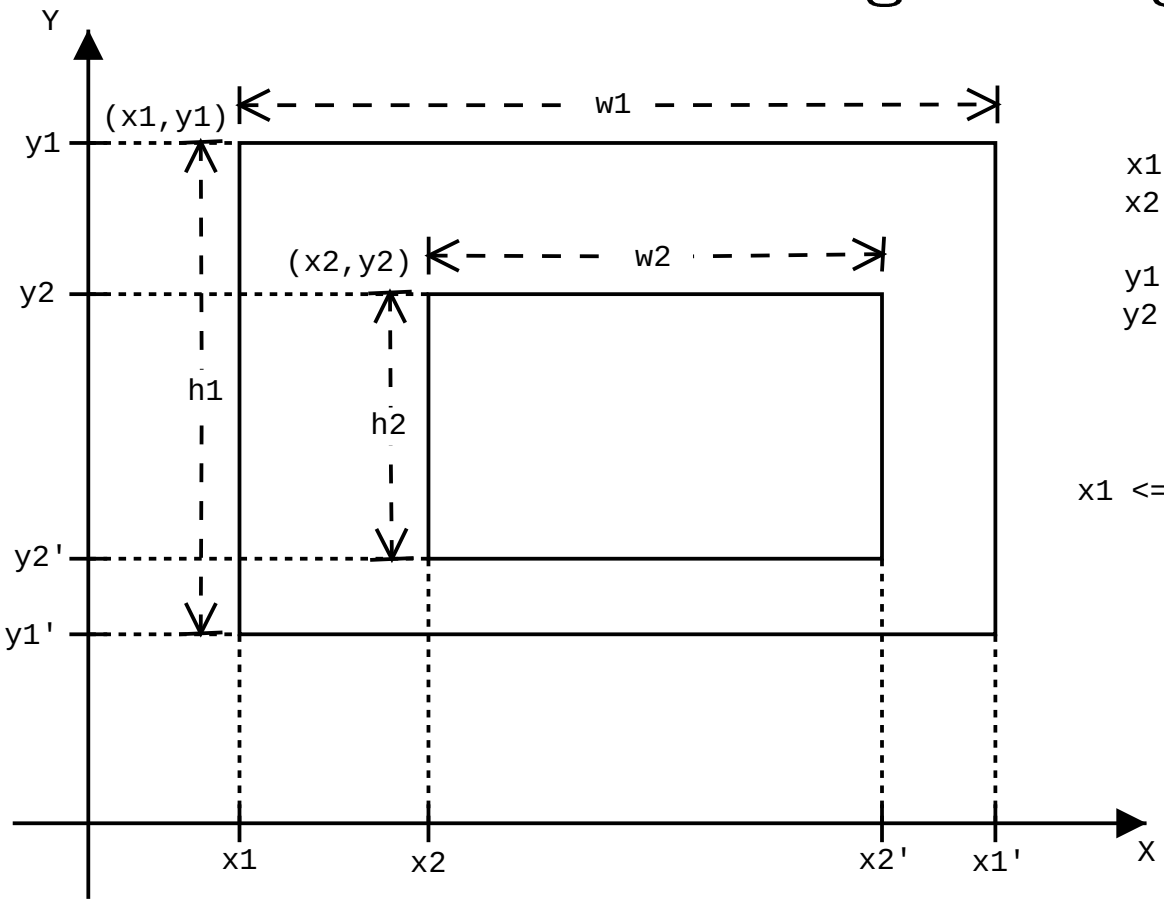
Section 3: Programming



Section 3: Programming



Section 3: Programming

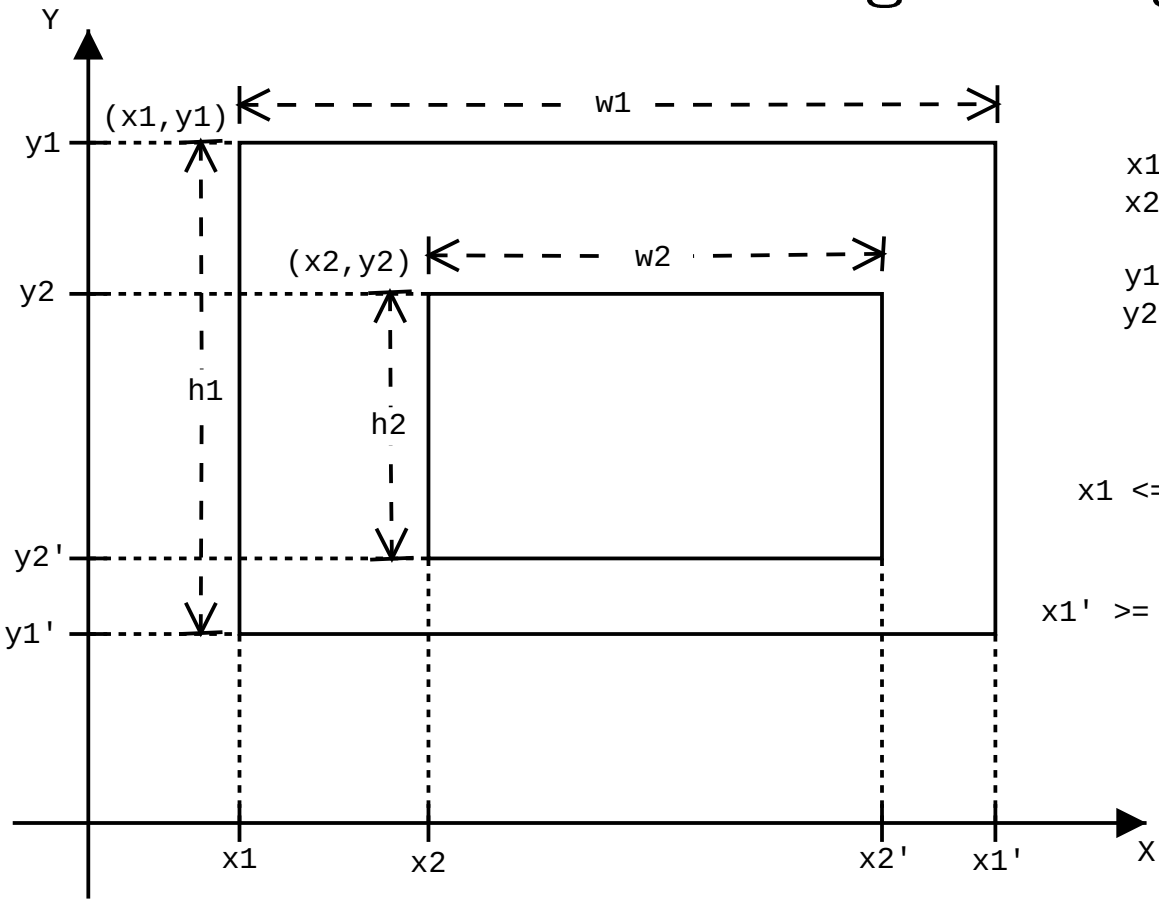



```
x1' = x1 + w1
x2' = x2 + w2

y1' = y1 - h1
y2' = y2 - h2
```

```
x1 <= x2   and   y1 >= y2
```

Section 3: Programming

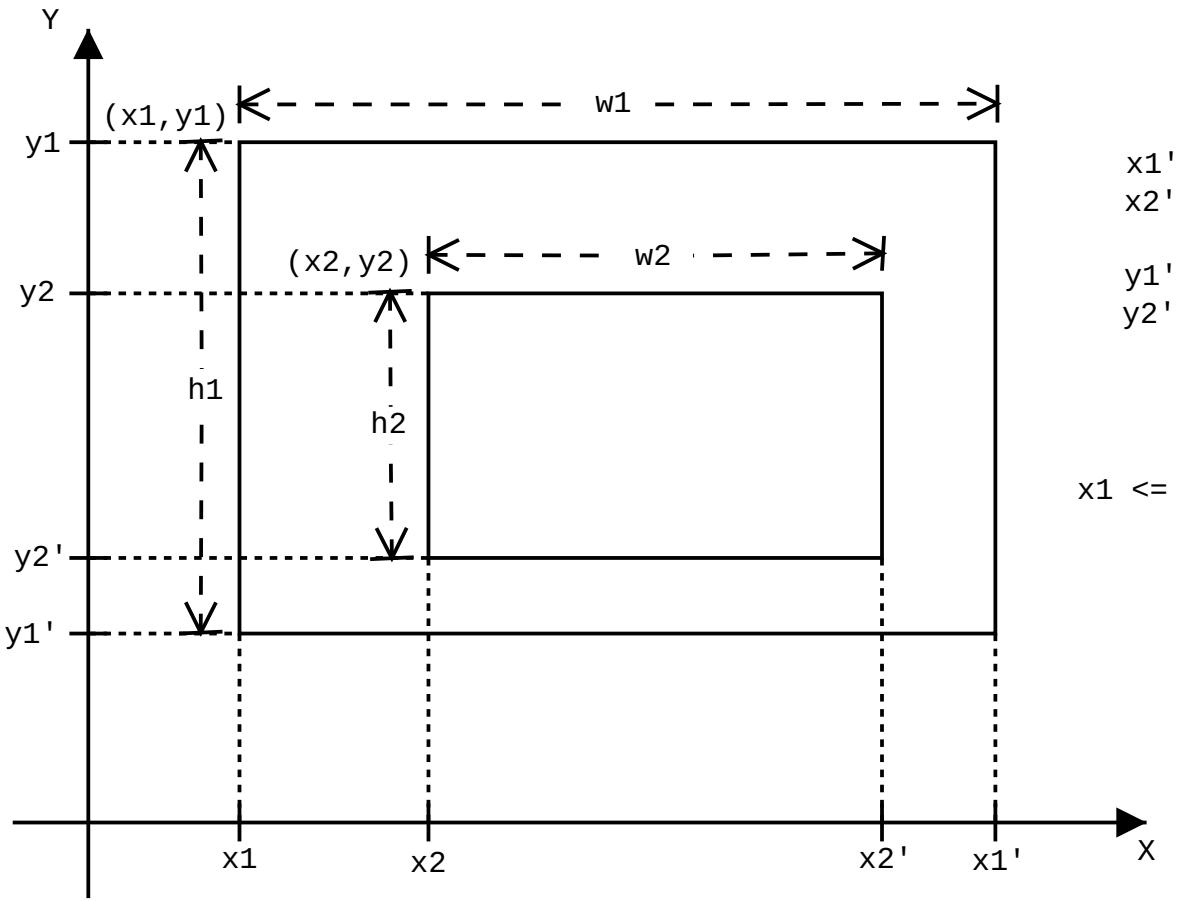



```
x1' = x1 + w1
x2' = x2 + w2

y1' = y1 - h1
y2' = y2 - h2
```

```
x1 <= x2    and    y1 >= y2
                   and
x1' >= x2'  and    y1' <= y2'
```

Section 3: Programming




```
x1' = x1 + w1
x2' = x2 + w2

y1' = y1 - h1
y2' = y2 - h2
```

```
x1 <= x2   and   y1 >= y2
           and
     x1+w1 >= x2+w2
           and
     y1-h1 <= y2-h2
```

Section 3: Programming

```
public class Rectangle {
  int x, y, width, height;

  Rectangle(int x, int y, int width, int height)
  {
    this.x = x;
    this.y = y;
    this.width  = width;
    this.height = height;
  }

  int area()
  {
    return width * height;
  }

  int getX()
  {
    return x;
  }
```

```
    int getY()
    {
      return y;
    }

    int getWidth()
    {
      return width;
    }

    int getHeight()
    {
      return height;
    }

    boolean contains(Rectangle r)
    {
      return this.x <= r.x
          && this.y <= r.y
          && this.x + width >= r.x + r.width
          && this.y - height <= r.y - r.height;
    }
  }
```

Recursive functions

Factorial defined recursively:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

This can be implemented as a static recursive method:

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

returns `(((factorial(0) * 1) * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

returns `(((factorial(0) * 1) * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

returns `(((1 * 1) * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

returns `(((1 * 1) * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

returns `((1 * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((factorial(1) * 2) * 3) * 4`

returns `((1 * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((1 * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `((1 * 2) * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `(2 * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(factorial(2) * 3) * 4`

returns `(2 * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(2 * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `(2 * 3) * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns 6 `* 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns `factorial(3) * 4`

returns `6 * 4`

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns 6 * 4

Recursion

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n - 1) * n;
}
```

...somewhere else ...

```
factorial(4)
```

returns 24

Recursion

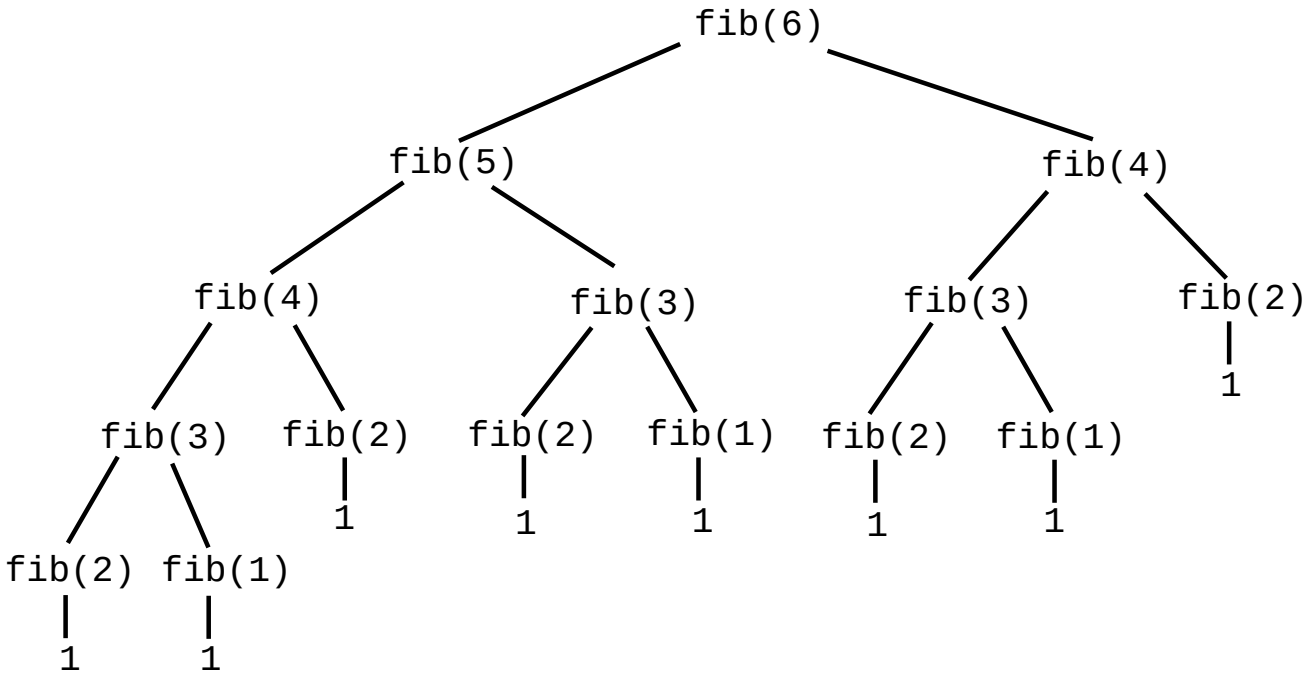
- Problem: Compute the n-th Fibonacci number
- Analysis: The Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ...is defined by:

$$fib(n) = \begin{cases} 1 & \text{if } n \leqslant 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Implementation:

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

Execution trees



Iteration vs recursion

- Iterative solution to the Fibonacci problem:

```
static int fib(int n)
{
    int a, b, c, i;
    a = 1;
    b = 1;
    c = 1;
    i = 3;
    while (i <= n) {
        c = a + b;
        a = b;
        b = c;
        i++;
    }
    return c;
}
```

Continuation-passing and tail recursion

```
static int factorial_2(int n)
{
  return cont_factorial(n, 1);
}

static int cont_factorial(int n, int result)
{
  if (n == 0) return result;
  return cont_factorial(n - 1, result * n);
}
```

Continuation-passing and tail recursion

```
factorial_2(4)
```

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 1 * 4);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 4 * 3);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 12 * 2);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns `cont_factorial(0, 24 * 1);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns `cont_factorial(0, 24);`

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns `cont_factorial(0, 24);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns `cont_factorial(1, 24);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns `cont_factorial(2, 12);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns `cont_factorial(3, 4);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns `cont_factorial(4, 1);`

returns 24

Continuation-passing and tail recursion

```
factorial_2(4)
```

returns 24

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2					
3					
4					

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6				
3	9				
4	12				

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6	8	10	12	14
3	9	17	27	39	53
4	12	29	56	95	148

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6	8	10	12	14
3	9	17	27	39	53
4	12	29	56	95	148

Recursion and termination

```
static void f(int n)
{
  System.out.println(n);
  f(n);
}
```

Recursion and termination

```
f(5)
f(5)
f(5)
f(5)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n) + 1;
}
```

Recursion and termination

```
f(5)
return f(5) + 1
return (f(5) + 1) + 1
return ((f(5) + 1) + 1) + 1
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return f(-1)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n);
}
```

Recursion and termination

```
f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
.
.
.
```

Recursion and termination

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

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return 1
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n - 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n / 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n + 2);
}
```

Recursion and termination

```
f(5)
returns f(7)
returns f(9)
returns f(11)
returns f(13)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( f( 3*n+1 ) );
}
```

Recursion and termination

Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geqslant 0 \\ A(m-1,1) & \text{if } m \geqslant 1 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m \geqslant 1 \text{ and } n \geqslant 1 \end{cases}$$

m\n	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125

Recursion and termination

Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geqslant 0 \\ A(m-1,1) & \text{if } m \geqslant 1 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m \geqslant 1 \text{ and } n \geqslant 1 \end{cases}$$

m\n	0	1	2	3	4	n
0	1	2	3	4	5	n+1
1	2	3	4	5	6	n+2
2	3	5	7	9	11	2n+3
3	5	13	29	61	125	$2^{(n+3)}$ –
4	13	65533	$2^{65536} - 3$			$2^{2^{\cdots^{2}}} - 3$ (n+

A(4,2) is greater than the number of particles in the universe raised to the power 200

A(5,2) cannot be written as a decimal expansion in the physical universe.

Recursion and termination

```
static int ackermann(int m, int n)
{
  if (m == 0 && n >= 0)
    return n + 1;
  else if (m >= 1 && n == 0)
    return ackermann(m - 1, 1);
  else
    return ackermann(m - 1, ackermann(m, n - 1));
}
```

The end