Recursão

Chamada Recursiva

- **Recursive call** **A method call in which the method being called is the same as the one making the call**
- **In other words, *recursion occurs when a method calls itself!***
- **Infinite recursion** **An infinite sequence of recursive method calls (not good)**

Finding a Recursive Solution

- A recursive solution to a problem must be written carefully
- The idea is for each successive recursive call to bring you one step closer to a situation in which the solution is known explicitly
- This situation in which the answer is known is called the base case
- Each recursive algorithm must have at least one base case, as well as a general (recursive) case

Forma geral de métodos recursivos

```
if (condição onde a solução é conhecida)  // Caso base
                                          // ou condição de parada
    retorno da solução

else                                      // General case
    chamada recursiva de método
```

Sum of numbers from 1 to n

DISCUSSION

The method call `summation(4)` (*somatório*) should return the value 10, because that is the result of

1 + 2 + 3 + 4 .

A situation where the answer is known is when n is 1: The sum of the numbers from 1 to 1 is certainly just 1.

So our base case could be along the lines of

```
if ( n == 1 )
        return 1;
```

Summing contd.

Now for the general case. . .

The sum of the numbers from 1 to n, that is,
1 + 2 + . . . + n, can be written as

n + the sum of the numbers from 1 to (n - 1),
that is, n + 1 + 2 + . . . + (n - 1)

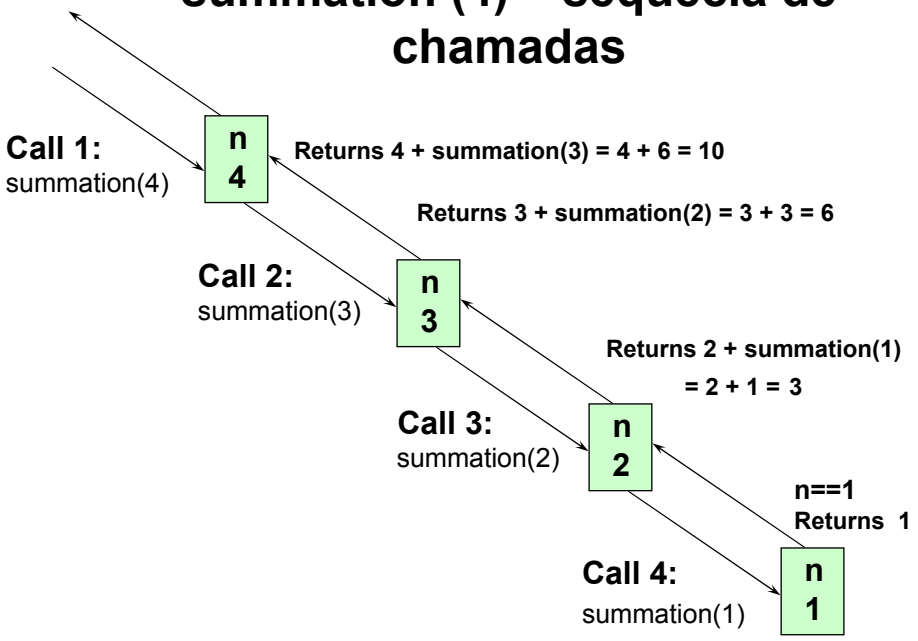
or, n + summation(n - 1)

And notice that the recursive call summation(n - 1) gets us "closer" to the base case of summation(1)

Code for summing numbers

```
public static int summation(int  n)
// Returns sum of numbers from 1 to n
// Assumption: n is greater than 0
// Computes the sum of the numbers from 1 to n by
// adding n to the sum of the numbers from 1 to
// (n-1)
{
  if (n == 1)                    // Base case
    return 1;
  else                           // General case
    return (n + summation(n - 1));
}
```

summation (4) – sequêcia de chamadas



A recursive n factorial

DISCUSSION

The method call `factorial(4)` should return the value 24, because that is the result of

4 * 3 * 2 * 1 .

For a situation in which the answer is known, the value of 0! is 1.

So our base case could be along the lines of

```
if ( number == 0 )
        return 1;
```

A recursive n factorial

Now for the general case . . .

The value of factorial(n) can be written as
n * the product of the numbers from (n - 1) to 1,
that is,

n * (n - 1) * . . . * 1

or, n * factorial(n - 1)

And notice that the recursive call factorial(n - 1) gets us "closer" to the base case of factorial(0).

Recursive Solution

```
public static int factorial(int  number)
// Assumption: number is greater than or equal to 0.
{
  if (number == 0)       // Base case
    return 1;
  else                   // General case
    return number * factorial (number - 1);
}
```

Another Natural Example

- From mathematics, we know that

  $2^0 = 1$ and $2^5 = 2 * 2^4$

- In general,

  $x^0 = 1$ and $x^n = x * x^{n-1}$

  for integer x, and integer n > 0.

- Here we are defining x^n recursively, in terms of x^{n-1}

```
// Recursive definition of power function
public static int power (int  x, int  n)
// Assumptions: n is greater than or equal to 0.
//              x and n are not both zero
// Returns x raised to the power n.
{
  if (n == 0)
    return 1;            // Base case
  else                   // General case
    return (x * power (x, n-1));
}
```

Of course, an alternative would have been to use looping instead of a recursive call in the method body

Extending the definition

- What is the value of 2^{-3} ? Again from mathematics, we know that it is

$$2^{-3} = 1 / 2^3 = 1 / 8$$

In general,

$$x^n = 1/ x^{-n}$$

for non-zero x, and integer n < 0.

Here we are again defining x^n recursively, in terms of x^{-n} when n < 0.

```
// Recursive definition of power function
public static double power(double x, int n)

// Returns x raised to the power n.
// Assumption: x is not zero and n has a value.

{
  if (n == 0)           // Base case
    return 1;
  else  if (n > 0)      // First general case
    return (x * power (x, n - 1));
  else                  // Second general case
    return (1.0 / power (x, - n))
}
```

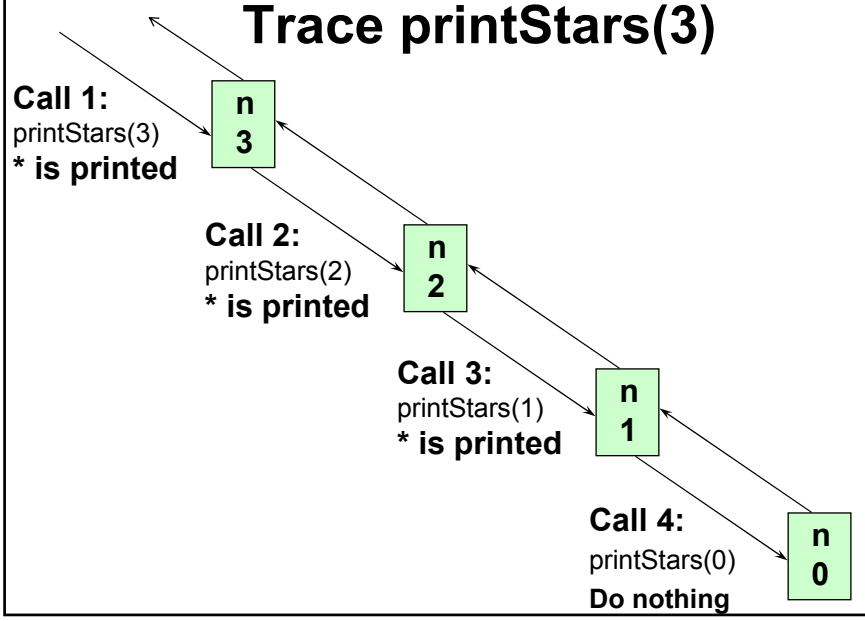
A do-nothing base case

```
public static void printStars(int  n)
// Assumption: n is greater than or equal to zero
// Result: n stars have been printed, one to a line
{
  if (n <= 0)           // Base case; Do nothing
  else                  // General case
  {
    outFile.println("*");
    printStars(n - 1);
  }
}
```

// CAN REWRITE AS . . .

```
public static void printStars(int n)
// Assumption: n is greater than or equal to zero
// Result: n stars have been printed, one to a line
{
  if (n > 0)                        // General case
  {
    outFile.println("*");
    printStars(n - 1);
  }
  // Base case is empty else-clause
}
```

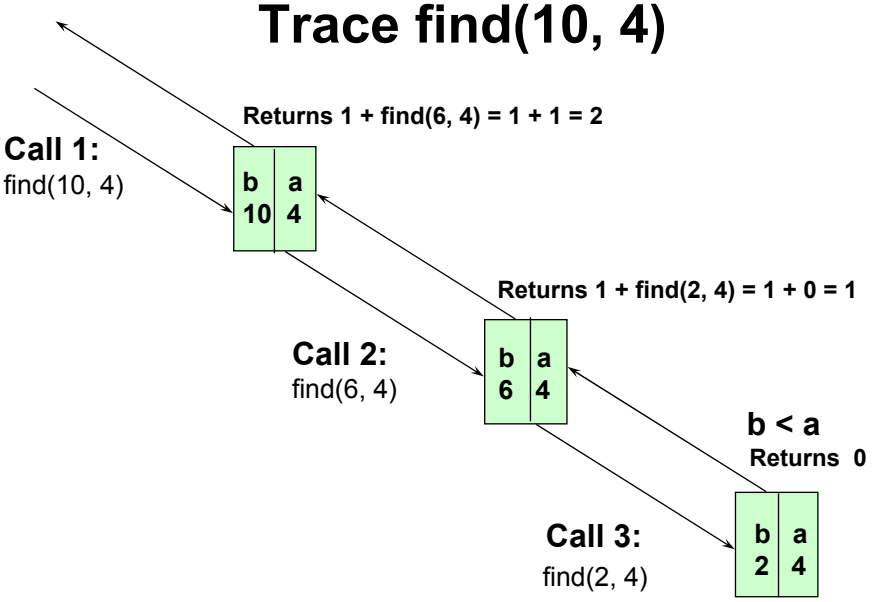
Trace printStars(3)



Recursive Mystery Function

```
public static int find(int b, int a)
// Simulates a familiar integer operator
// Assumption: a is greater than zero and
//             b is greater than or equal to zero
// Returns   ???
{
  if (b < a)              // Base case
    return 0;
  else                    // General case
    return (1 + find (b - a, a));
}
```

Trace find(10, 4)

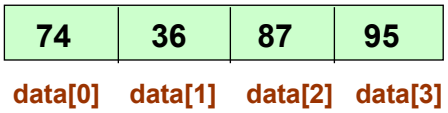


Print items in reverse order

DISCUSSION

For this task, we will use the heading:

```
public static void printRev(int[] data, int first,
  int last)
```



The call

```
printRev (data, 0, 3);
```

should produce this output: 95 87 36 74

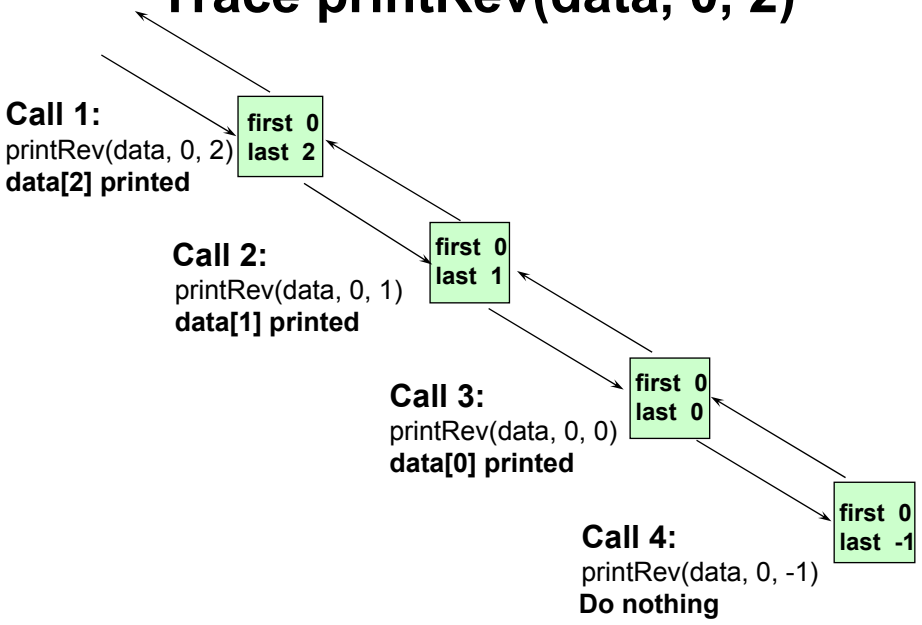
Base Case and General Case

- **A base case may be a solution in terms of a "smaller" array; certainly for an array with 0 elements, there is no more processing to do**
- **Now our general case needs to bring us closer to the base case situation; that is, the length of the array to be processed decreases by 1 with each recursive call.**
- **By printing one element in the general case, and also processing the smaller array, we will eventually reach the situation where 0 array elements are left to be processed**
- **In the general case, we could print the rest of the array and then the first element, or we could print the last element and then the rest of the array; let's do the latter**

Using recursion with arrays

```
public static void printRev
   (int[] data,     // Array to be printed
    int first,      // Index of first element
    int last)       // Index of last element
// Prints data[first]...data[last] in reverse order
{
  if (first <= last)     // General case
  { outFile.print(data[last] + "    ");
    // print last element
    printRev (data, first, last - 1); // print rest
  }
  // Base case is empty else-clause
}
```

Trace printRev(data, 0, 2)



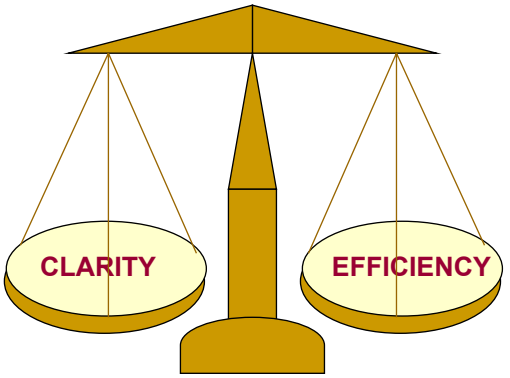
Por que usar Recursão?

- These examples could all have been written without recursion, by using iteration instead; the iterative solution uses a loop, and the recursive solution uses a selection statement
- However, for certain problems the recursive solution is the most natural solution; this often occurs when structured variables are used

Recall that . . .

- Recursion occurs when a method calls itself (directly or indirectly)
- Selection constructs are used in recursion, not looping constructs
- Recursion is a natural choice for some algorithms; iteration is a natural choice for others

Recursion or Iteration?



What is the value of `rose(25)`?

```
public static int rose(int n)
{
  if (n == 1)   // Base case
    return  0;
  else          // General case
    return (1 + rose (n / 2));
}
```

Finding the value of `rose(25)`

	rose(25)	*the original call*
=	1 + rose(12)	*first recursive call*
=	1 + (1 + rose(6))	*second recursive call*
=	1 + (1 + (1 + rose(3)))	*third recursive call*
=	1 + (1 + (1 + (1 + rose(1))))	*fourth recursive call*
=	1 + 1 + 1 + 1 + 0	
=	4	

Writing recursive functions

- There must be at least one base case, and at least one general (recursive) case
- The general case should bring you "closer" to the base case
- The values in the recursive call cannot all be the same as the values in the original call; otherwise, infinite recursion would occur.
- In method `rose()` , the base case occurred when `(n == 1)` was true; the general case brought us a step closer to the base case, because the argument in the general case was `rose(n/2)`, which was closer to 1, than `n` was.

When a method is called...

- Control is transferred from the calling block to the code of the method
- After the method is executed, control must be returned to the proper place in the calling code, which is called the return address
- When any method is called, an activation record (or stack frame) for the method call is placed on the run-time stack

Stack Activation Frames

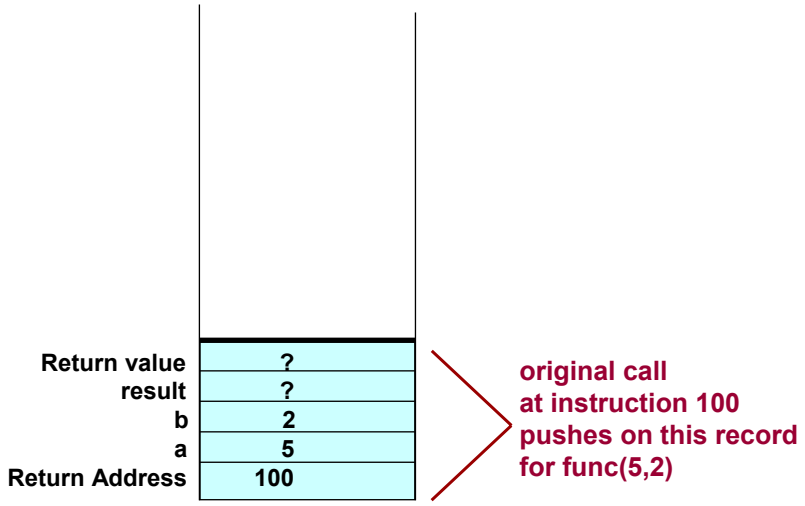
- The activation record contains the return address for this method call, the arguments, space for local variables, and space for a non-void method's return value
- The activation record for a particular method call is removed from the run-time stack when the final closing brace or a return statement is reached in the method's code
- At this time a value-returning method's return value is sent back to the calling block for use there

Another Recursive Example

```
// Another recursive method
public static int func(int a, int b)
// Returns  ??
{
  int result;
  if (b == 0)                 // Base case
    result = 0;
  else if (b > 0)            // First general case
    result = a + func(a, b - 1));
    // instruction 50
  else                        // Second general case
    result = func(- a, - b);
    // instruction 70
  return result;
}
```

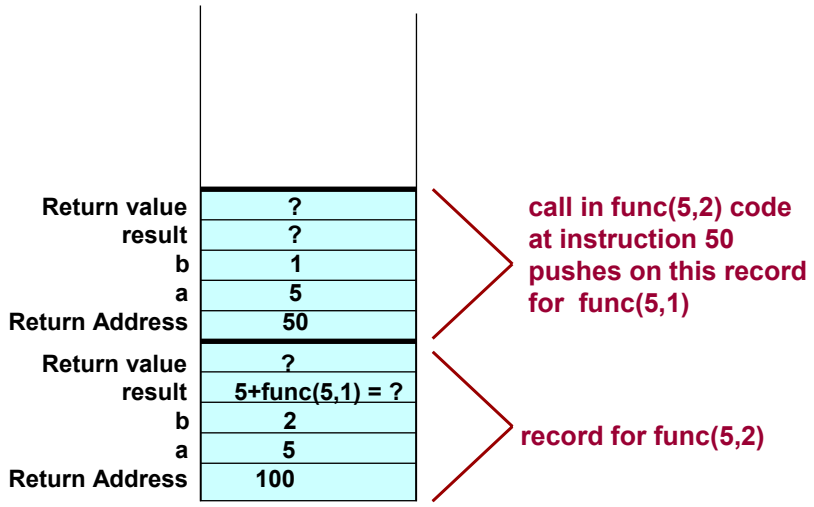
Activation Records on Stack

`x = func(5, 2);` // original call at instruction 100



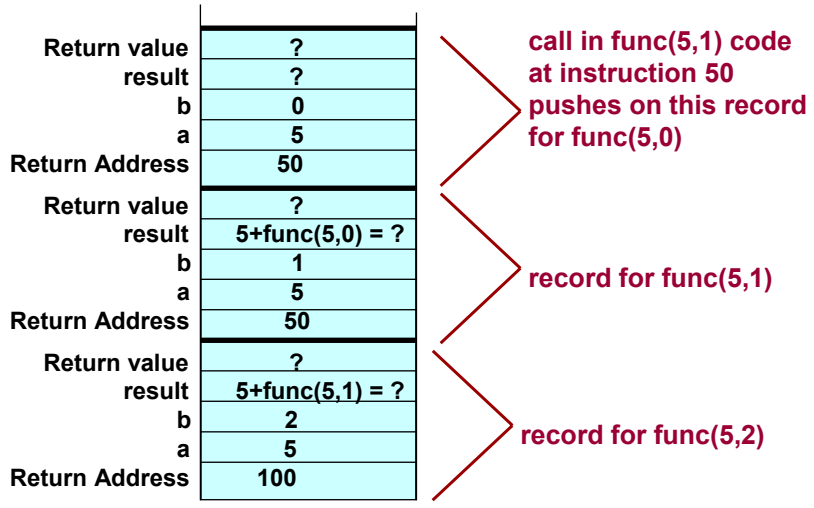
Activation Records on Stack

```
x = func(5, 2);          // original call at instruction 100
```



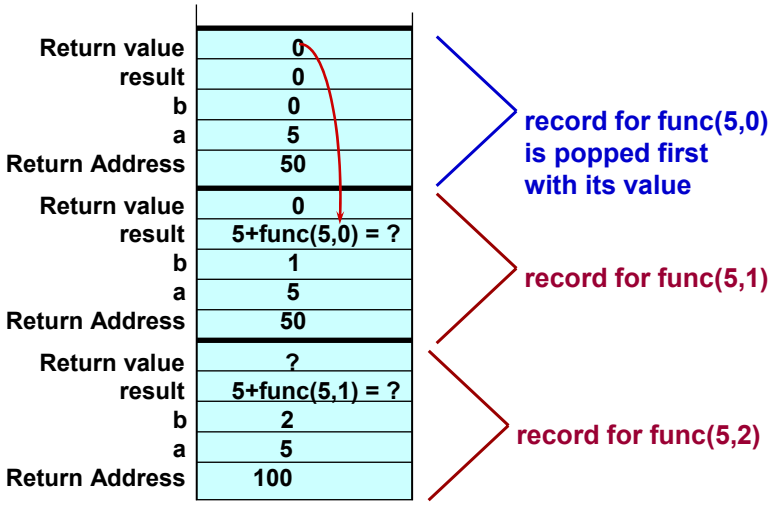
Activation Records on Stack

```
x = func(5, 2);          // original call at instruction 100
```



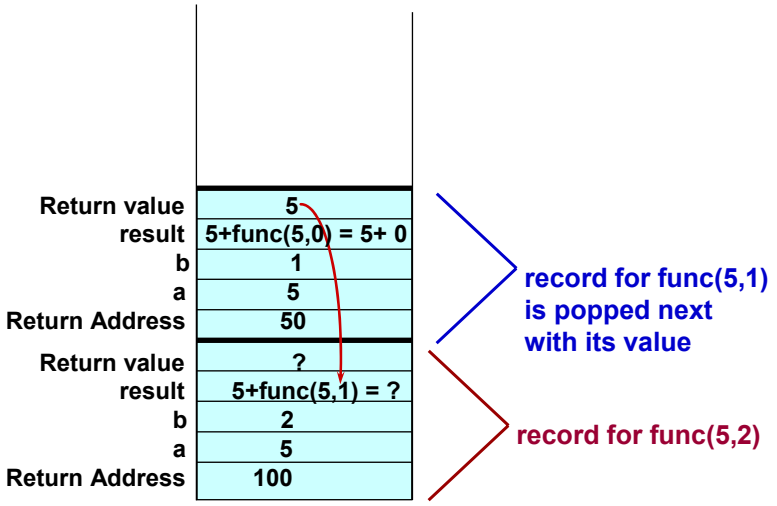
Activation Records on Stack

```
x = func(5, 2);          // original call at instruction 100
```



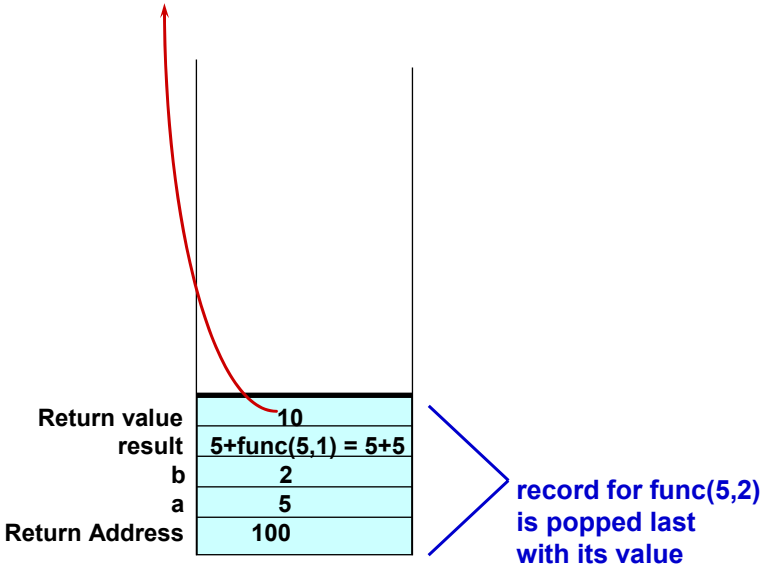
Activation Records on Stack

```
x = func(5, 2);          // original call at instruction 100
```



Activation Records on Stack

```
x = func(5, 2);
```
// original call at instruction 100



Trace these Calls

```
x = func(- 5, - 3);

x = func(5, - 3);
```

What operation does `func(a, b)` simulate?

Write a method . . .

- `sum` that takes an array `a` and two subscripts, `low` and `last` as parameters, and returns the sum of the elements `a[low] + . . . + a[last]`
- Write the method two ways -- using iteration and using recursion
- *For your recursive definition's base case, for what kind of array do you know the value of `sum(a, low, last)` right away*?

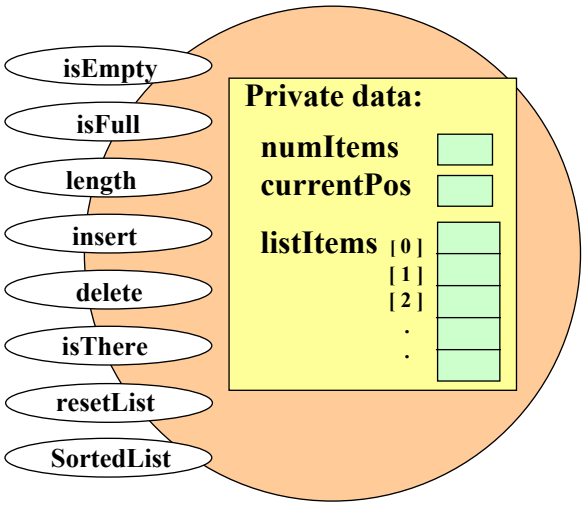
```
// Recursive definition
public static int sum(int[] a,
                      int low,
                      int last)

// Returns sum of a[low] . . . a[last]

{
  if (low == last)       // Base case
    return  a[low];
  else                    // General case
    return a[low] + sum(a, low + 1, last);
}
```

Write a method . . .

- `linearSearch` that takes an array `a` and two subscripts, `low` and `high`, and `item` as parameters.
- Returns true if `item` is found in the elements `a[low]...a[high]`; otherwise, returns false.
- Write the method using recursion
- *For your base case(s), for what kinds of arrays do you know the value of* `linearSearch(a, low, high, item)` *right away?*

```
// Recursive sequential search
public static int linearSearch
  (int[] a, int low, int high, int item)
// Returns true if item is in a[low]…a[high];
// false otherwise.
{
  if (a[low] == item)          // First base case
    return true;
  else  if (low == high)       // Second base case
    return false;
  else                         // General case
    return linearSearch(a, low+1, high, item);
}
```

Array-based `class SortedList`



Remember Binary Search?

- Examines the element in the middle of the array.
  - *Is it the sought item?*
    - If so, stop searching
  - *Is the middle element too small?*
    - Look in second half of array
  - *Is the middle element too large?*
    - Look in first half of the array
- Repeat the process in the half of the array that should be examined next
- Stop when item is found or when there is nowhere else to look and item has not been found

Binary Search Algorithm

- The Binary Search algorithm is,
  - Divide the list in half and decide which half to look in next
  - Repeat division of the selected portion until the item is found or it is determined that the item is not in the list
- This algorithm is recursive!

Recursive Binary Search

- The Binary Search algorithm can be written using iteration or recursion
- `binIsThere` takes **sorted** array `listItems`, and two subscripts, `first` and `last`, and `item` as parameters
- It returns true if `item` is found in the elements `listItems[first]...listItems[last];` otherwise, it returns false.

```
located = binIsThere(0, 14, 25);
```

first last item

subscripts

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	2	4	6	8	10	12	(14)	16	18	20	22	24	26	28

listItems

16 18 20 (22) 24 26 28

24 (26) 28

(24)

NOTE: ( ) denotes element examined

```
// Recursive definition

private boolean binIsThere(int first, int last,
  int item)
// Assumption: List items are in ascending order
// Returns true if item is found; false otherwise
{
  if (first > last)      // Base case 1 -- not found
    return false;
  else
  { int mid;
    mid = (first + last) / 2;
    if (listItems[mid] == item)
    // Base case 2 -- found at mid
      return  true;
    else  if (item < listItems[mid])
    // search lower half
      return binIsThere(first, mid - 1, item);
    else // search upper half
      return  binIsThere(mid + 1, last, item);}
}
```

Recursive Binary Search

- The recursive binary search function must be called from the `isThere` method of `SortedList` class.

```
public boolean isThere(int item)
// Returns true if item is in the list;
// false otherwise
// Assumption: List items are in ascending
// order
{
   return binIsThere(0, numItems - 1, item);
}
```

Write a method . . .

- `minimum` that takes an array `a` and the `size` of the array as parameters, and returns the smallest element of the array; that is, it returns the smallest value of `a[0] ... a[size-1]`
- Write the method two ways -- using iteration and using recursion
- For your recursive definition's base case, for what kind of array do you know the value of `minimum(a, size)` right away?

```
// Recursive definition
public static int minimum(int[] a, int size)
// Returns  smallest of a[0]...a[size - 1]
{
  if (size == 1)  // Base case
    return  a[0];
  else
  {// General case
    int y = minimum(a, size - 1);
    if (y < a[size - 1])
      return y;
    else
       return a[size -1];
  }
}
```