Data Structures & Algorithms

Stack & Recursion

NUST

SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Today‘s lecture

- Stacks
  - Practical applications
    - Stacks in validating expressions
    - **Infix to Postfix conversion & evaluation**
      - Algorithms implementing stack
        - To convert any Infix expression into its corresponding Postfix expression
        - To evaluate any postfix expression
      - Postfix/Prefix forms do not rely on operator priorities, a tie breaker,
        or delimiters
        - Easier evaluation of expressions
        - Unambiguous expression tree
    - Adding very large integer numbers
- Recursion

Another stack application example: Adding very large integers

- Consider adding very large numbers
  - Since the largest magnitude of integers is limited (32 bits), so we are not able to perform following:

18,274,364,583,929,273,748,459,595,684,373

+

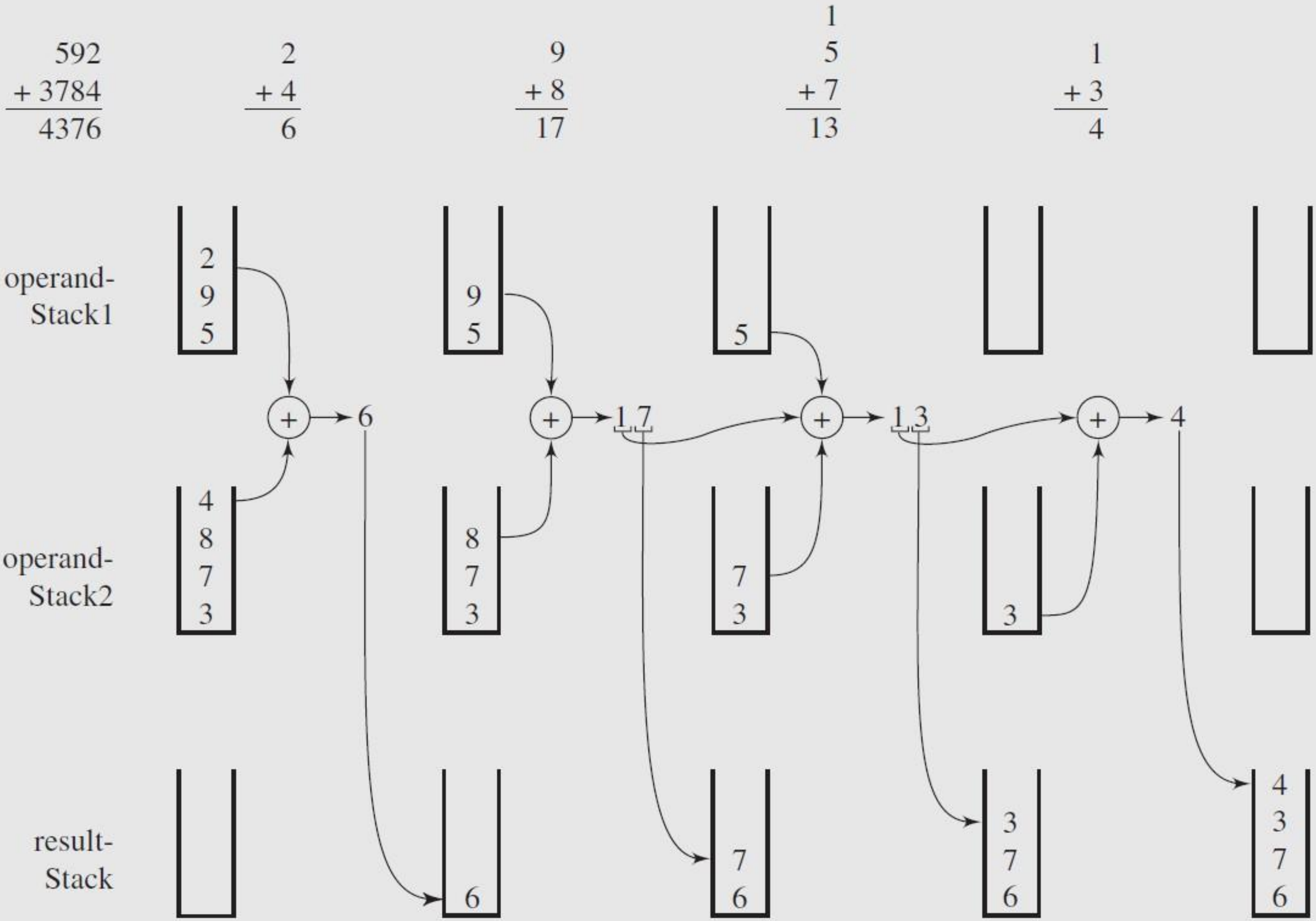
8,129,498,165,026,350,236

because integer variables cannot hold such large values

- The problem can be solved if we
  - treat these numbers as strings of numerals,
  - store the numbers corresponding to these numerals on two stacks, and
  - then perform addition by popping numbers from the stacks

Pseudocode

```
addingLargeNumbers()
    Read the numerals of the first number and store the numbers corresponding to them on one stack;
    Read the numerals of the second number and store the numbers corresponding to them on another stack;
    carry = 0;
    while at least one stack is not empty
        - pop a number from each nonempty stack and add them to carry;
        - push the unit part on the result stack;
        - store carry in carry;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```


592
+ 3784
4376
2
+ 4
6
9
+ 8
17
1
5
+ 7
13
1
+ 3
4
operand-
Stack1
operand-
Stack2
result-
Stack
2
9
5
4
8
7
3
+
6
17
13
4
376

Today‘s lecture

- Stacks
  - Practical applications
    - Stacks in validating expressions
    - **Infix to Postfix conversion & evaluation**
      - Algorithms implementing stack
        - To convert any Infix expression into its corresponding Postfix expression
        - To evaluate any postfix expression
      - Postfix/Prefix forms do not rely on operator priorities, a tie breaker,
        or delimiters
        - Easier evaluation of expressions
        - Unambiguous expression tree
    - Adding very large integer numbers
- Recursion

Recursive thinking

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that would be difficult to solve in other ways

- Recursion splits a problem into one or more simpler versions of itself
  - These sub-problems may also be divided into further smaller sub-sub-problems

- A recursive call is a function call in which the called function is the same as the one making the call

Recursive Definitions

- Recursive algorithm
  - Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself
  - Has one or more base cases
  - Implemented using recursive functions

- Recursive function
  - Function that calls itself

Recursive Definitions

- Base case
  - Case in recursive definition in which the solution is obtained directly
  - Stops the recursion

- General case
  - Case in recursive definition in which a smaller version of itself is called
  - Must eventually be reduced to a base case

Outline of a recursive function

```
if (answer is known)
    provide the answer
```

Base case

```
else
    make a recursive call to
    solve a smaller version
    of the same problem
```

Recursive case

Tracing a recursive function

- Recursive function
  - May have unlimited copies of itself
  - Every recursive call has
    - its own code
    - own set of parameters
    - own set of local variables

- After completing recursive call
  - Control goes back to calling environment
  - Recursive call must execute completely before control goes back to previous call
  - Execution in previous call begins from point immediately following recursive call

How Recursion Works?

- Recursive function call is handled like any other function call

- Each recursive call has an activation record or a stack frame
  - Stores values of parameters and local variables

- When base case is reached, return is made to previous call
  - Recursion “unwinds”

Factorial definition

4! = 4 * 3 * 2 * 1

n! = n * (n-1) * … * 1

4! = 4 * 3!

n! = n * (n-1)!

factorial(n) = n * factorial(n-1) // recursive step

factorial(1) = 1 // base step

$$n! = \left\{\begin{array}{ll} 1 & if n = 0 \\ n * (n - 1\,!) & if\ n > 0 \end{array}\right\}$$

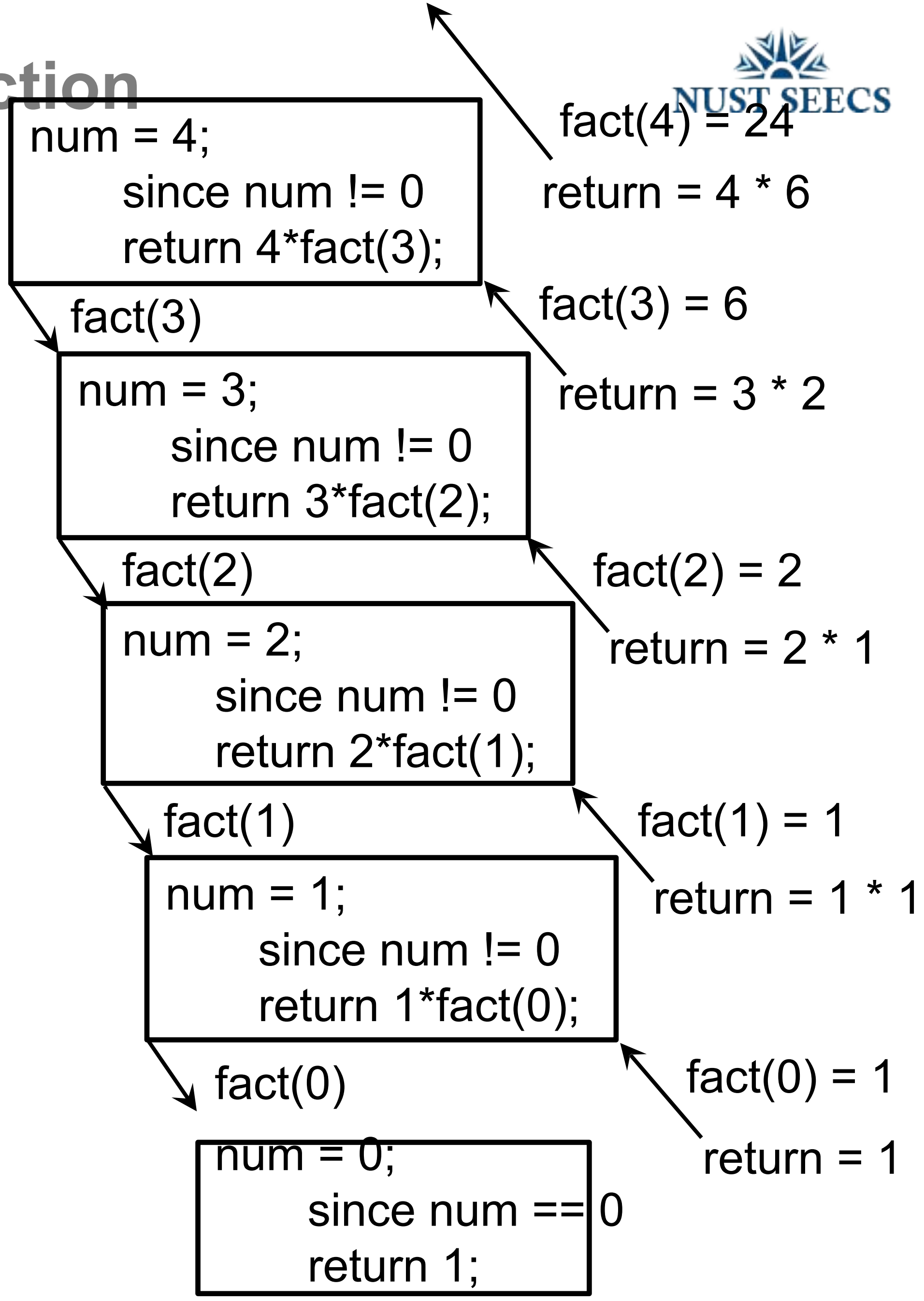
Recursive factorial function

```
int fact(int num)
{
   if(num == 0)
      return 1;
   else
      return num * fact(num – 1);
}
```

Recursive factorial function

```
int fact(int num)
{
   if(num == 0)
      return 1;
   else
      return num * fact(num – 1);
}
```

Recursive factorial trace



num = 4;
since num != 0
return 4*fact(3);

fact(3)

num = 3;
since num != 0
return 3*fact(2);

fact(2)

num = 2;
since num != 0
return 2*fact(1);

fact(1)

num = 1;
since num != 0
return 1*fact(0);

fact(0)

num = 0;
since num == 0
return 1;

fact(0) = 1
return = 1

fact(1) = 1
return = 1 * 1

fact(2) = 2
return = 2 * 1

fact(3) = 6
return = 3 * 2

fact(4) = 24
return = 4 * 6

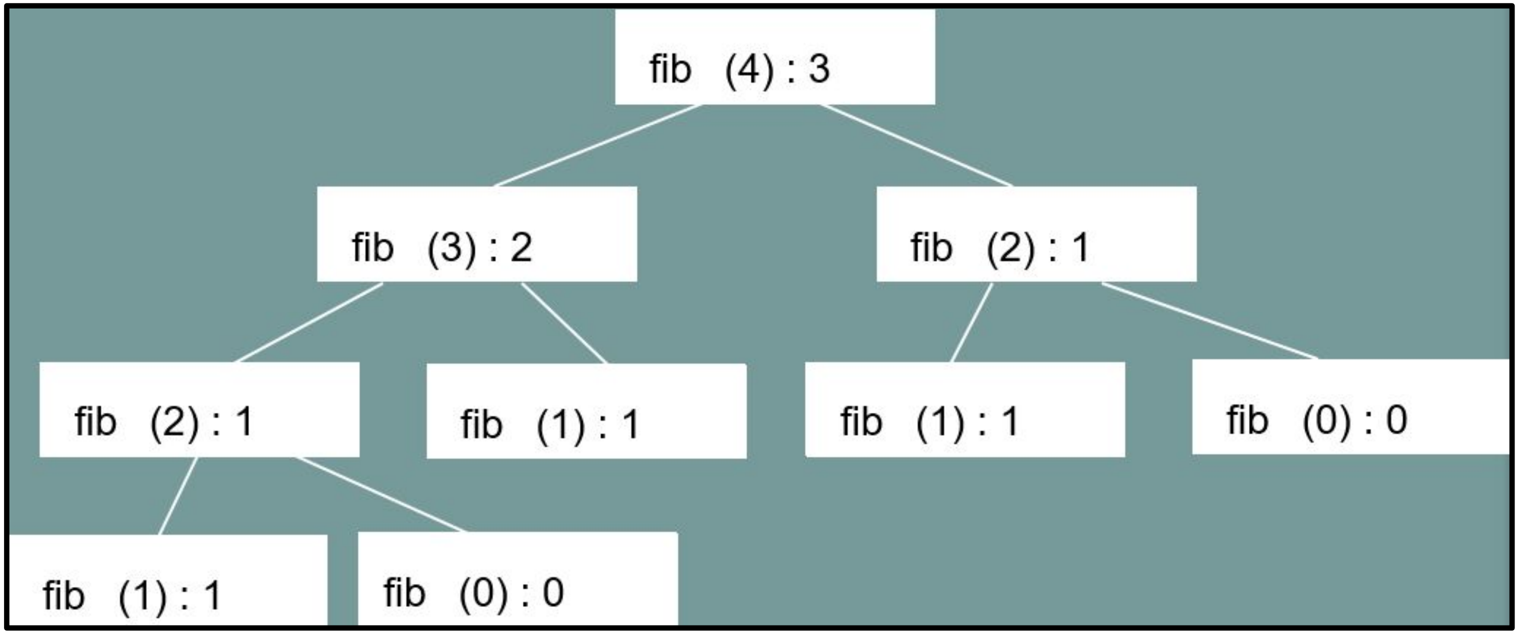
Another recursive definition

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$fib(n) = \begin{cases} 0 & for\ n == 0 \\ 1 & for\ n == 1 \\ fib(n-2) + fib(n-1) & for\ n >= 2 \end{cases}$$

fib(n) = fib(n-1) + fib(n-2)	**Recursive case**
fib(0) = 0 fib(1) = 1	**Base case**

Recursive Fibonacci repeats computations



```
int fib (int n)
{
   if (n == 0) {
      return 0; }
   else if (n == 1) {

      return 1; }
   else {
      return fib (n-1) + fib (n-2); }
 }
```

Evaluating Exponents Recursively

```
double power (double x, unsigned int n) // raise x to the power n
{
if (n == 0)
        return 1.0;
    else
        return x * power(x,n-1);
}
```

Using this definition, the value of x^4 can be computed in the following way:

$$x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) \quad = x \cdot (x \cdot (x \cdot x^1)) \quad = x \cdot (x \cdot (x \cdot (x \cdot x^0)))$$

$$= x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x))$$

$$= x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$$

Implementation using loop

The function power() can also be implemented differently, without using any recursion, as in the following loop:

```
double nonRecPower(double x, unsigned int n) {
double result = 1;
for (result = x; n > 1; --n)
        result *= x;
        return result;
}
```

Do we gain anything by using recursion instead of a loop?

Recursion or iteration?

- The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition
- The recursive version increases program readability, improves self-documentation, and simplifies coding
- In the example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the non-recursive implementations

Recursive vs iteration

- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop

- In recursion, the condition usually tests for a base case

- You can always write an iterative solution to a problem that is solvable by recursion

- Recursive code may be simpler than an iterative algorithm and thus easier to write, read, and debug

When to use recursion?

- If recursive and iterative algorithms are of similar efficiency
  - Prefer iteration over recursion

- If the problem is inherently recursive and a recursive algorithm is less complex to write than an iterative algorithm
  - Prefer recursion over iteration

Efficiency of recursion

- Recursive methods often have slower execution times when compared to their iterative counterparts
  - Because the overhead for loop repetition is smaller than the overhead for a method call and return

- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
  - Because sometimes the reduction in efficiency does not outweigh the advantage of readable code that is easy to debug

Pitfalls of recursion

- One pitfall of recursion is infinite regress, i.e. a chain of recursive calls that never stops
  - E.g. if you forget the base case. Make sure you have enough base cases

- Recursion can be less efficient than an iterative implementation. This can happen because there is recalculation of intermediate results

- There can be extra memory use because recursive calls must be stored on the execution stack

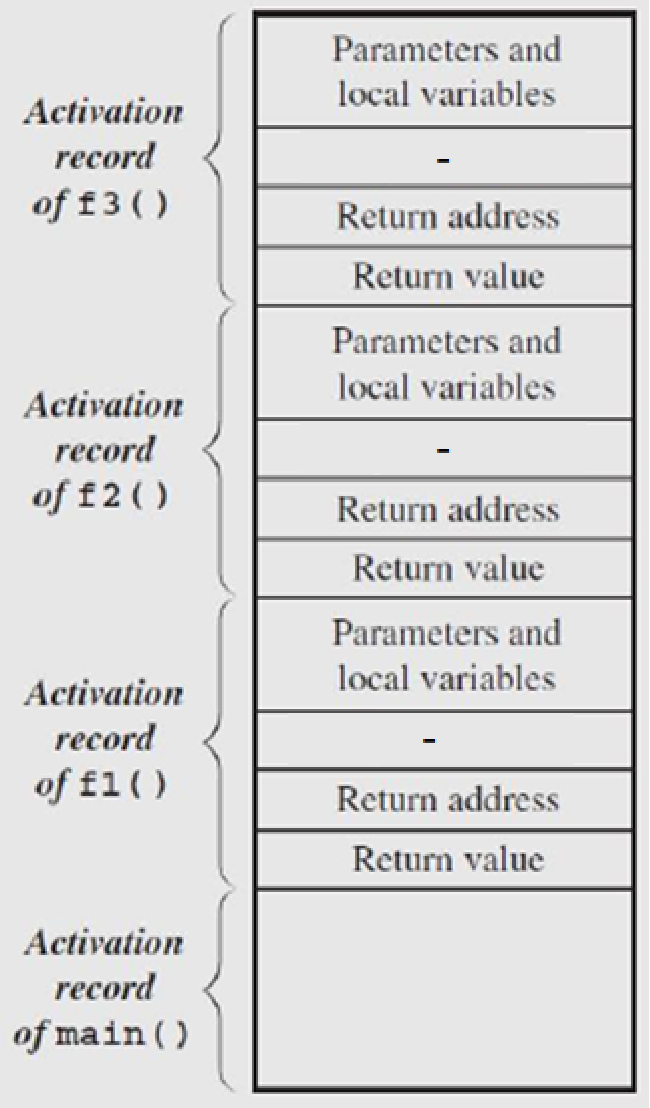
Function calls & recursive implementation

- The state of each function, including main(), is characterized
  - by the contents of all local variables,
  - by the values of the function’s parameters, and
  - by the return address indicating where to restart in the calling function
- The data area containing all this information is called an activation record or a stack frame and is allocated on the run-time stack
  - exists for as long as a function owning it is executing
- This record is a private pool of information for the function, a repository that stores all information necessary for its proper execution and how to return to where it was called from

Activation Record / Stack Frame

- If a function is called either by main() or by another function, then its stack frame is created on the run-time stack

- The run-time stack always reflects the current state of the function
  - E.g., suppose that main() calls function f1(), f1() calls f2(), and f2() in turn calls f3(). If f3() is being executed, then the state of the run-time stack is as shown


Activation record of f3()
Parameters and local variables
-
Return address
Return value
Activation record of f2()
Parameters and local variables
-
Return address
Return value
Activation record of f1()
Parameters and local variables
-
Return address
Return value
Activation record of main()

Activation Record / Stack Frame

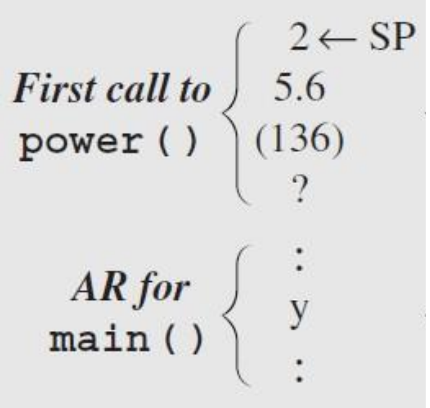
- An activation record usually contains the following information
  - Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items;
  - Local variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored;
  - The return address to resume control by the caller, the address of the caller’s instruction immediately following the call;
  - The returned value for a function not declared as void. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller

Evaluating Exponents Recursively

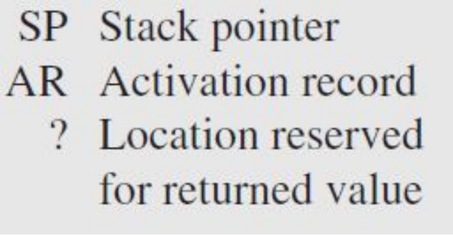
```
double power (double x, unsigned int n) /* 102 */
{
if (n == 0)                              /* 103 */
        return 1.0;                      /* 104 */
    else
        return x * power(x,n-1);         /* 105 */
}

int main()
{ ...
             y = power(5.6,2);           /* 136 */
...
}
```

First, the value of the second argument, i.e., 2, is checked, and power() tries to return the value of 5.6 * power(5.6,1)

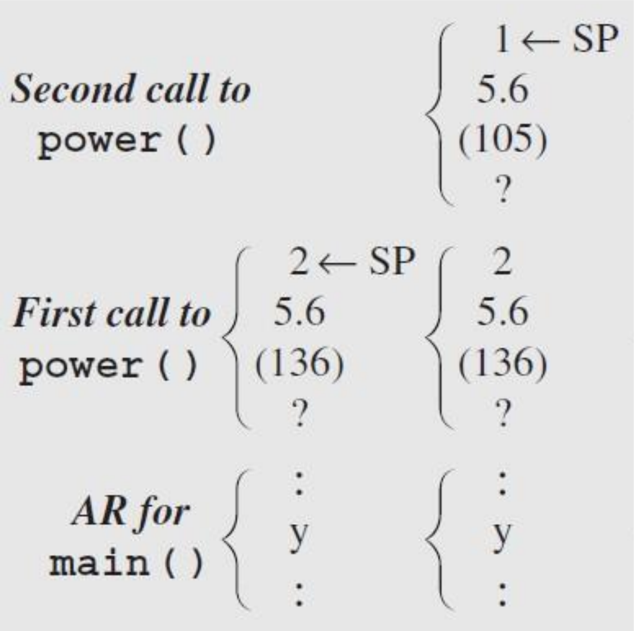


```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
    return 1.0;                           /* 104 */
else
    return x * power(x,n-1); }            /* 105 */
```

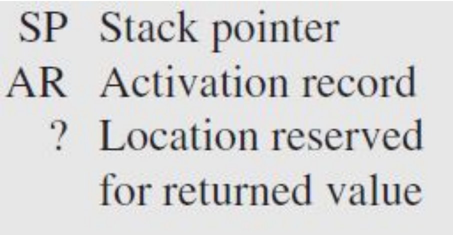


SP	Stack pointer
AR	Activation record
?	Location reserved for returned value

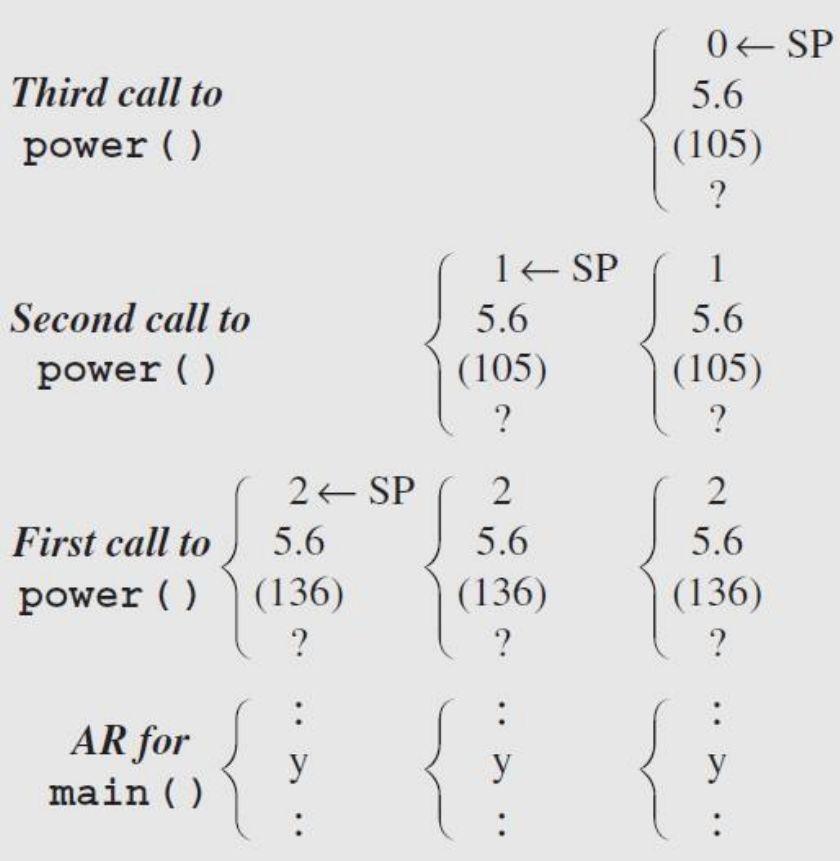
Again first, the value of the second argument, i.e., 1, is checked, and power() tries to return the value of 5.6 * power(5.6,0)



```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
      return 1.0;                         /* 104 */
else
      return x * power(x,n-1); }          /* 105 */
```

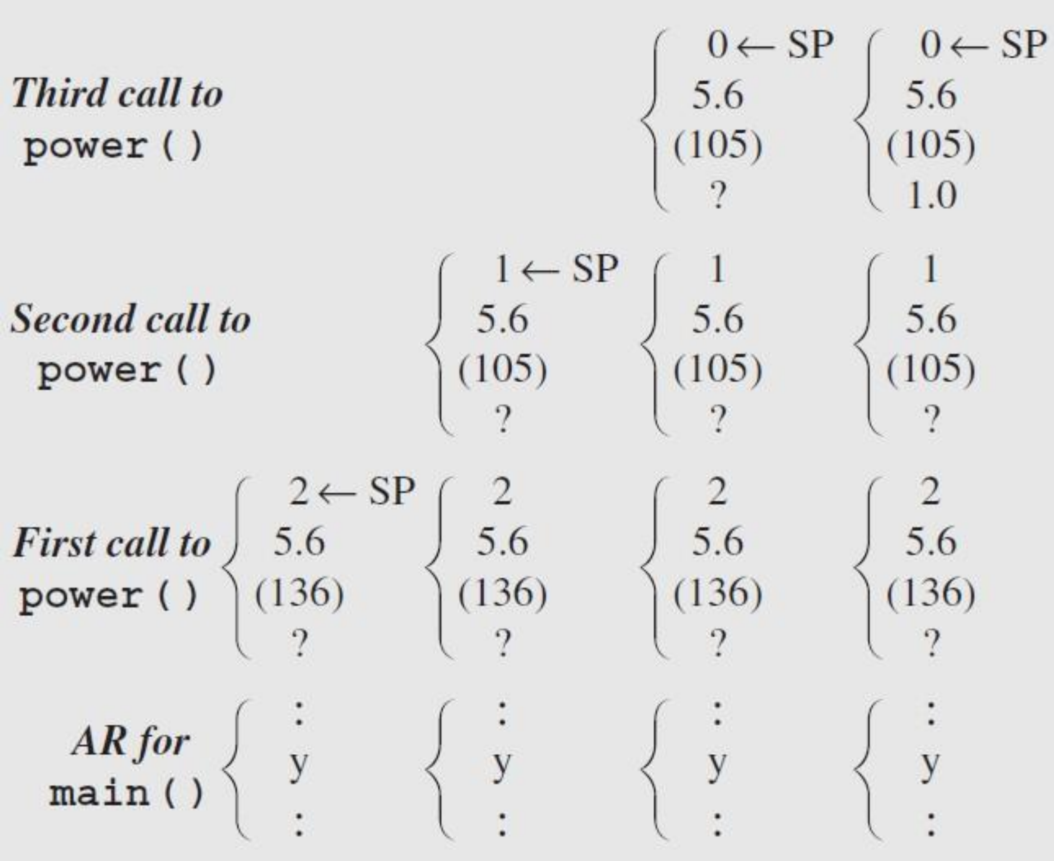


SP	Stack pointer
AR	Activation record
?	Location reserved for returned value



```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
      return 1.0;                         /* 104 */
else
      return x * power(x,n-1); }          /* 105 */
```

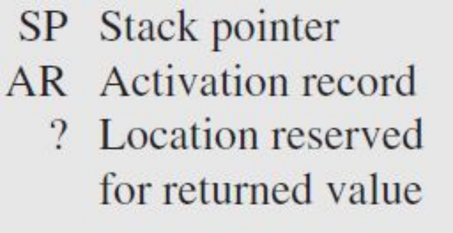
SP	Stack pointer
AR	Activation record
?	Location reserved for returned value



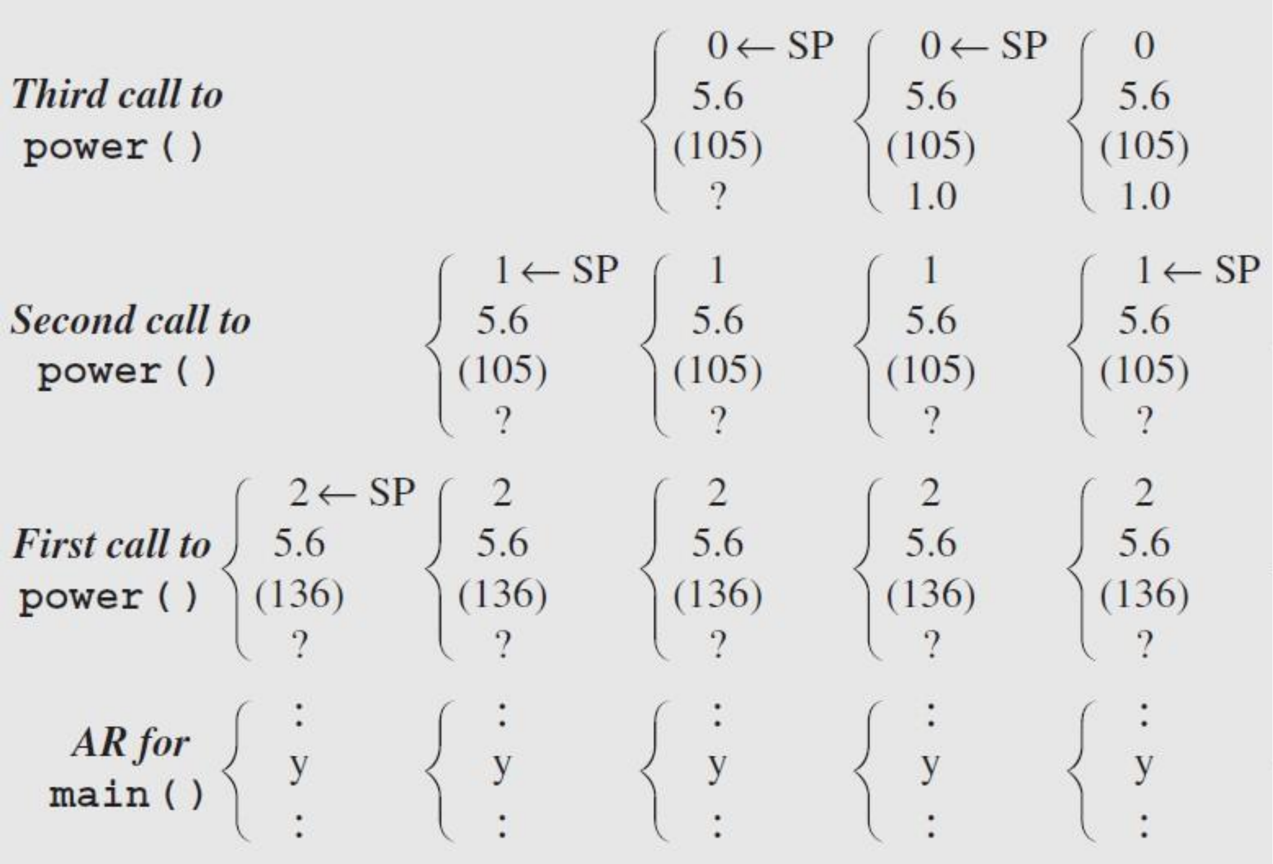
Again, first, the value of the second argument, i.e., 0, is checked, and power() returns the value 1

At this point, there are two pending calls on the run-time stack—the calls to power()—that have to be completed

```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
      return 1.0;                         /* 104 */
else
      return x * power(x,n-1); }          /* 105 */
```

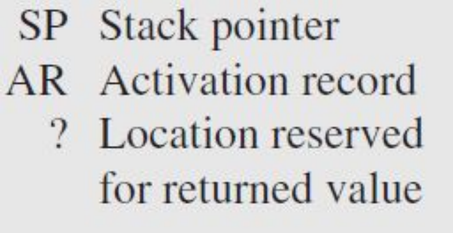


SP	Stack pointer
AR	Activation record
?	Location reserved for returned value

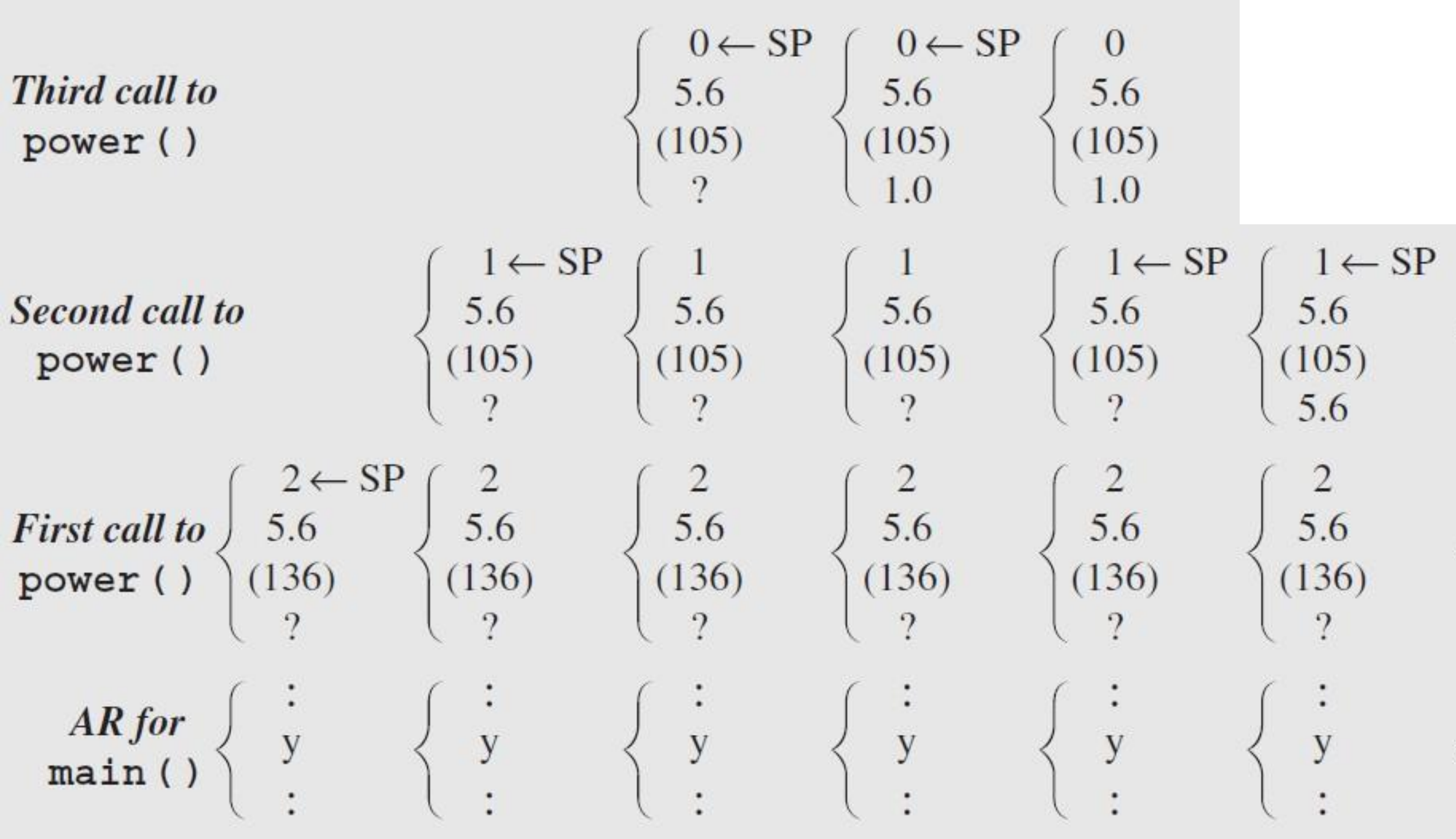


5.6 * power(5.6,0) = 5.6 * 1 = 5.6 is computed

```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
      return 1.0;                         /* 104 */
else
      return x * power(x,n-1); }          /* 105 */
```



SP	Stack pointer
AR	Activation record
?	Location reserved for returned value

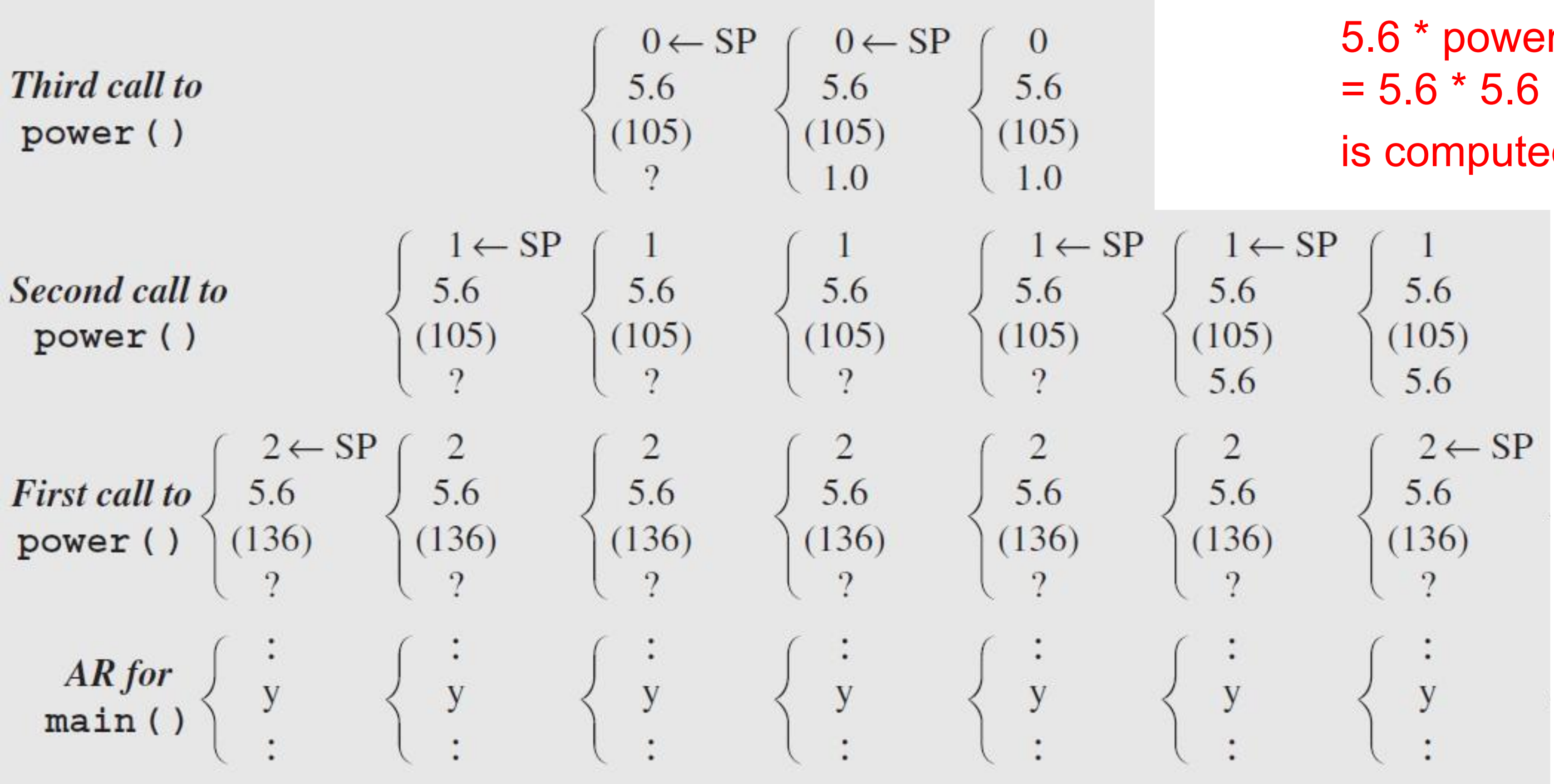


Third call to `power()`			0 ← SP 5.6 (105) ?	0 ← SP 5.6 (105) 1.0	0 5.6 (105) 1.0	
Second call to `power()`		1 ← SP 5.6 (105) ?	1 5.6 (105) ?	1 5.6 (105) ?	1 ← SP 5.6 (105) ?	1 ← SP 5.6 (105) 5.6
First call to `power()`	2 ← SP 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?
AR for `main()`	: y :	: y :	: y :	: y :	: y :	: y :

5.6 * power(5.6,0) = 5.6 * 1 = 5.6 is computed

```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
      return 1.0;                         /* 104 */
else
      return x * power(x,n-1); }          /* 105 */
```

SP	Stack pointer
AR	Activation record
?	Location reserved for returned value

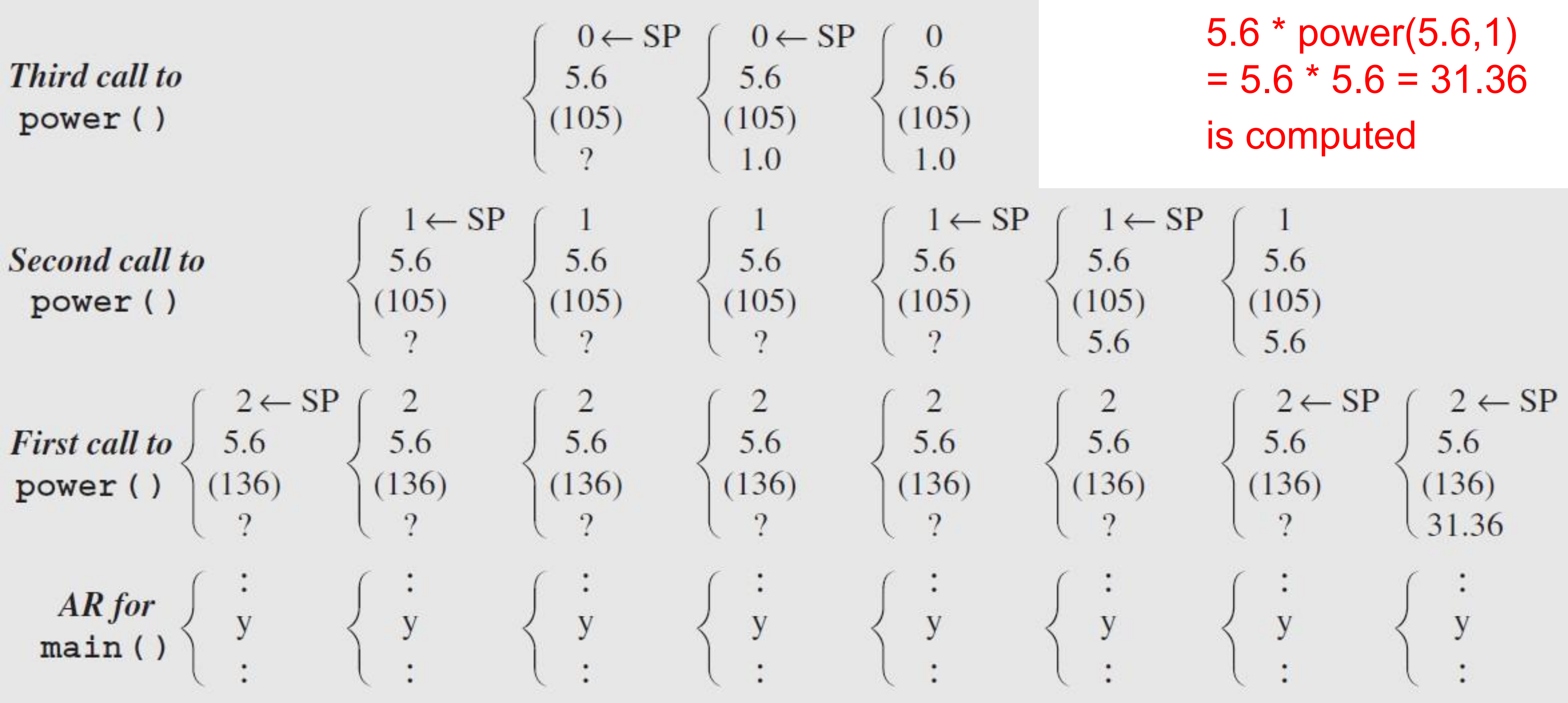


Third call to power()			0 ← SP 5.6 (105) ?	0 ← SP 5.6 (105) 1.0	0 5.6 (105) 1.0		
Second call to power()		1 ← SP 5.6 (105) ?	1 5.6 (105) ?	1 5.6 (105) ?	1 ← SP 5.6 (105) ?	1 ← SP 5.6 (105) 5.6	1 5.6 (105) 5.6
First call to power()	2 ← SP 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 ← SP 5.6 (136) ?
AR for main()	: y :	: y :	: y :	: y :	: y :	: y :	: y :

5.6 * power(5.6,1)
= 5.6 * 5.6 = 31.36
is computed

```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
      return 1.0;                         /* 104 */
else
      return x * power(x,n-1); }          /* 105 */
```

SP	Stack pointer
AR	Activation record
?	Location reserved for returned value



Third call to power()			0 ← SP 5.6 (105) ?	0 ← SP 5.6 (105) 1.0	0 5.6 (105) 1.0			
Second call to power()		1 ← SP 5.6 (105) ?	1 5.6 (105) ?	1 5.6 (105) ?	1 ← SP 5.6 (105) ?	1 ← SP 5.6 (105) 5.6	1 5.6 (105) 5.6	
First call to power()	2 ← SP 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 5.6 (136) ?	2 ← SP 5.6 (136) ?	2 ← SP 5.6 (136) 31.36
AR for main()	: y :	: y :	: y :	: y :	: y :	: y :	: y :	: y :

5.6 * power(5.6,1)
= 5.6 * 5.6 = 31.36
is computed

```
double power (double x, unsigned int n) { /* 102 */
if (n == 0)                               /* 103 */
        return 1.0;                       /* 104 */
else
        return x * power(x,n-1); }        /* 105 */
```

SP	Stack pointer
AR	Activation record
?	Location reserved for returned value