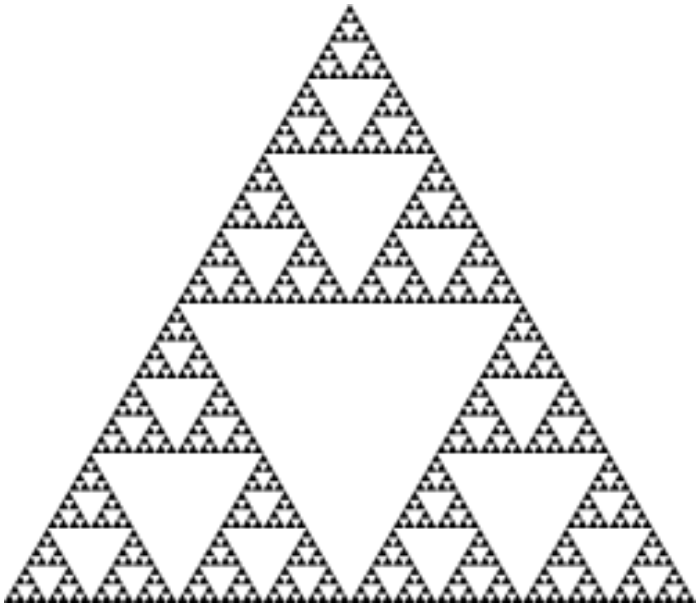
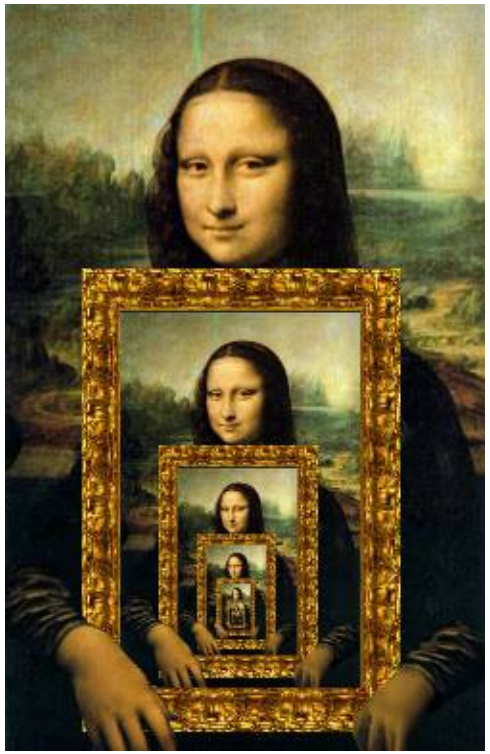
Chapter 17: Recursion

Recursion

- Recursion, is a process of repeating something in a self-similar way (also known as self-similarity)





Recursion in Computer Science

- *Recursion* is an alternative to *iteration*.
  - It is **repetition** without control of a loop.

1. A function calls itself.
2. A base case determines whether or not to make the recursive call.
   - e.g. there's no more data to look at
3. The recursive call is assigned an argument that is moving toward the base case.
   - e.g. smaller amount of data

Iterative Example: Count

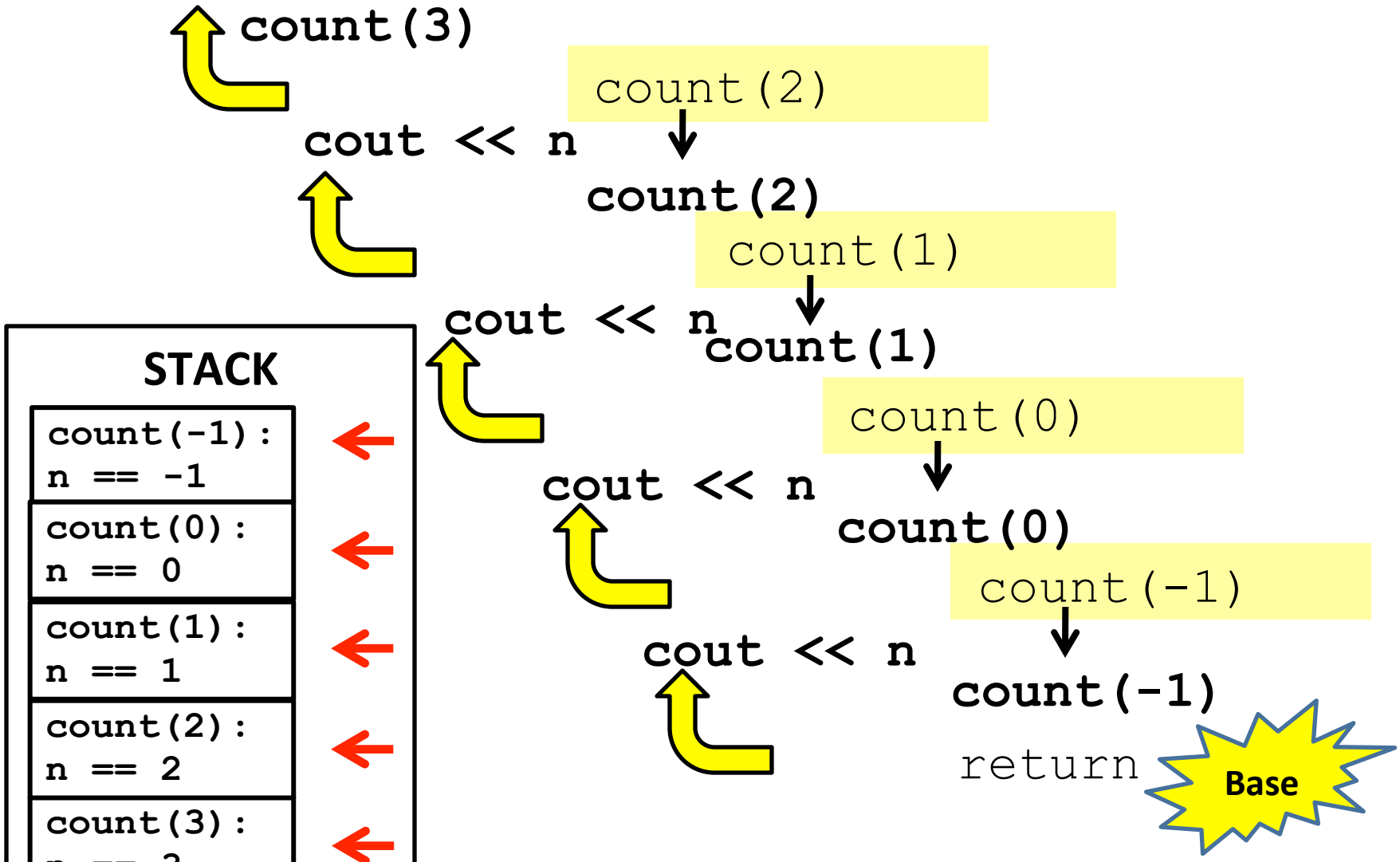
```
void count(int n) {
    for (int i = 0; i <= 10; i++)
        cout << i << endl;
}
```

Recursion Example

1. Function **calls itself**
2. A **base case** determines whether or not to make the recursive call.
3. The recursive call **argument is smaller**

```
void count( int n )
{
      if ( n <= -1 )
            return ;
      else {
            count( n - 1 );
            cout << n << endl;
      }
}
```

Animation of Recursive Count



Factorial

1. **The factorial of integer *n* is the product of all positive integers less than or equal to *n***

2. **The factorial of 0 is 1**

$$n\,! = n * (n\text{-}1) * (n\text{-}2) * \ldots * 1$$
$$0! = 1$$

OR

$$n! = n * (n\text{-}1)!$$
$$0! = 1$$

Recursive implementation of Factorial.

```
// input n must be >= 0
double factorial( int n )
{
      if ( n == 0 )
            return 1.0;
      else
            return n * factorial( n - 1 );
}
```

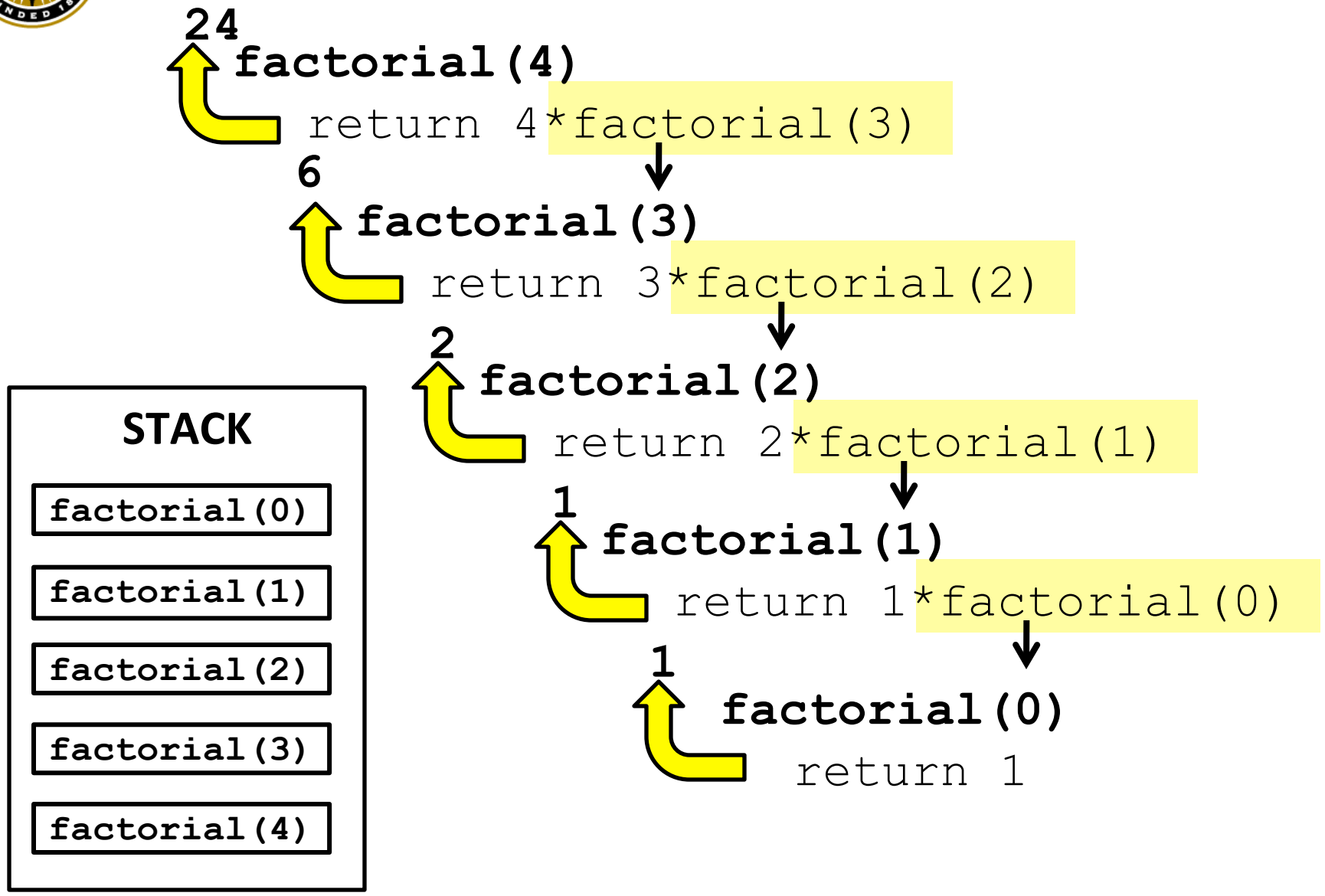
Programming Factorial Recursively

```
#include <iostream>
using namespace std;

  double factorial( int n )
  {
        if ( n == 0 )
              return 1.0;
        else
              return n * factorial( n - 1 );
  }

  int main()
  {
        cout << factorial(4) << endl;
  }
```

Animation of Recursive Factorial



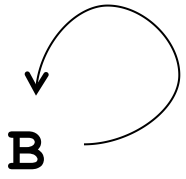
In-class Exercise

- Try this yourself

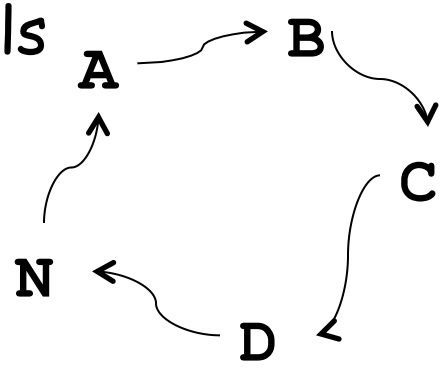
- Implement and test count() and factorial()
  - Make sure they work
  - Make sure you understand them
- Implement sum()
  - Should be very similar to factorial
- Challenge
  - Write a recursive function that takes a vector of ints and determines if the vector is sorted
  - Recursive definition: a vector is sorted if
    1. it has only one item, or if it has 0 items, or if the current item is greater than it's following item
    2. if the current item is less than it's following item, and if the rest of the vector is sorted

Direct and Indirect Recursion

- Direct recursion
  - a function B calls repeatedly itself



- Indirect recursion
  - Function A calls B, and function B calls function A. Or,
  - Function A calls B, which calls C …, which calls function A.



Fibonacci numbers can be generated recursively

fib(0) = 0;

fib(1) = 1;

fib(i) = fib(i-1) + fib(i -2)

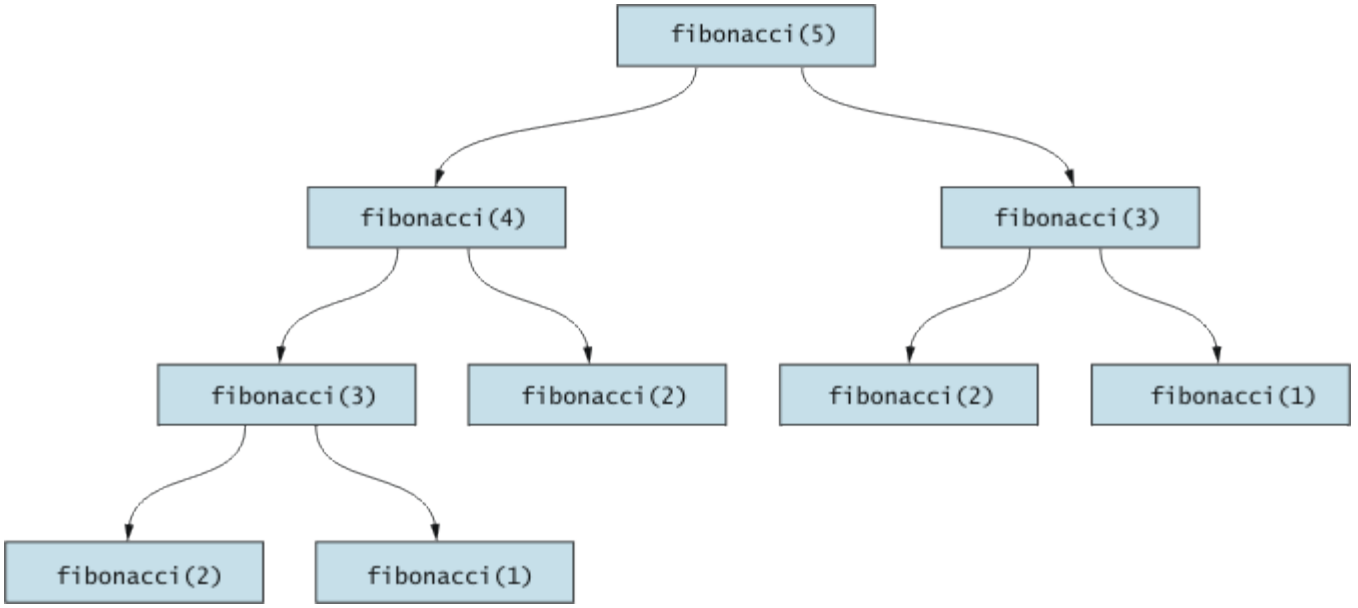
```
fib(3) = fib(2) + fib(1)
       = (fib(1) + fib(0)) + fib(1)
       = (1 + 0) +fib(1)
       = 1 + fib(1) = 1 + 1 = 2
```

```
Finonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…
         indices: 0 1 2 3 4 5 6 7  8  9  10 11
```

Recursive Fibonacci Implementation

```
// The function for finding the Fibonacci number
double fibonacci(int index)
{
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1.0;
  else // Reduction and recursive calls
    return fibonacci(index - 1) +
           fibonacci(index - 2);
}
```

Efficiency of Recursion: Inefficient Fibonacci



Iterative Fibonacci implementation

```
double fibonacci ( int N )
{
      double prev = 0; // fib(0)
      double curr = 1.0; // fib(1)
      for (int i = 2; i <= N; i++) {
                  double temp = curr + prev;
            prev = curr;
            curr = temp;
      }
      return curr;
}
```

Recursion vs. Iteration

Recursion:

- Natural formulation of solution to certain problems
- Results in shorter, simpler functions
- May not execute efficiently (except in some cases)

Iteration:

- Generally executes more efficiently than recursion
- May not be as natural as recursion for some problems (e.g. Fibonacci)

Problem:

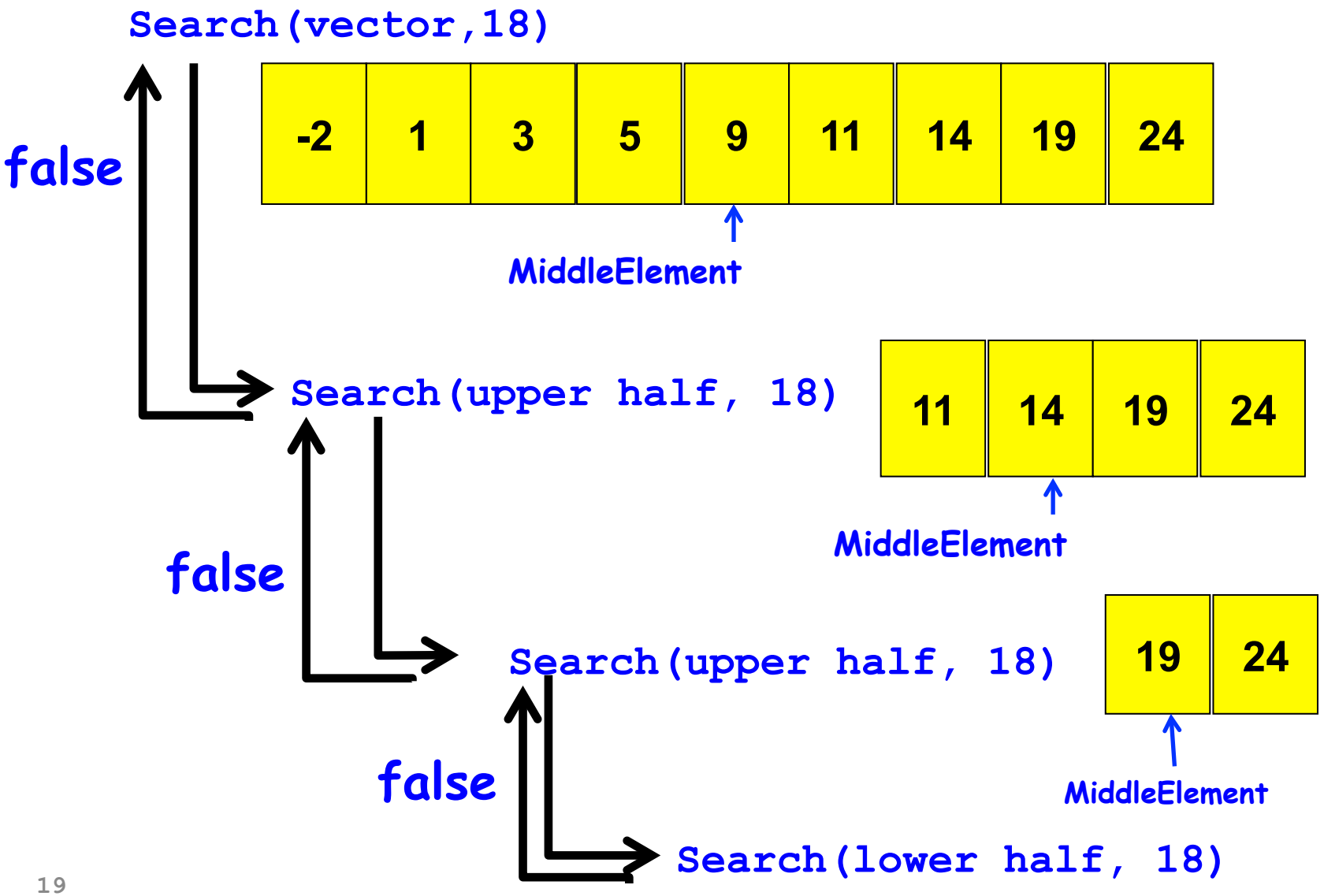
How do you **efficiently** determine if an item is in a **sorted vector**?

Is "18" in this vector?

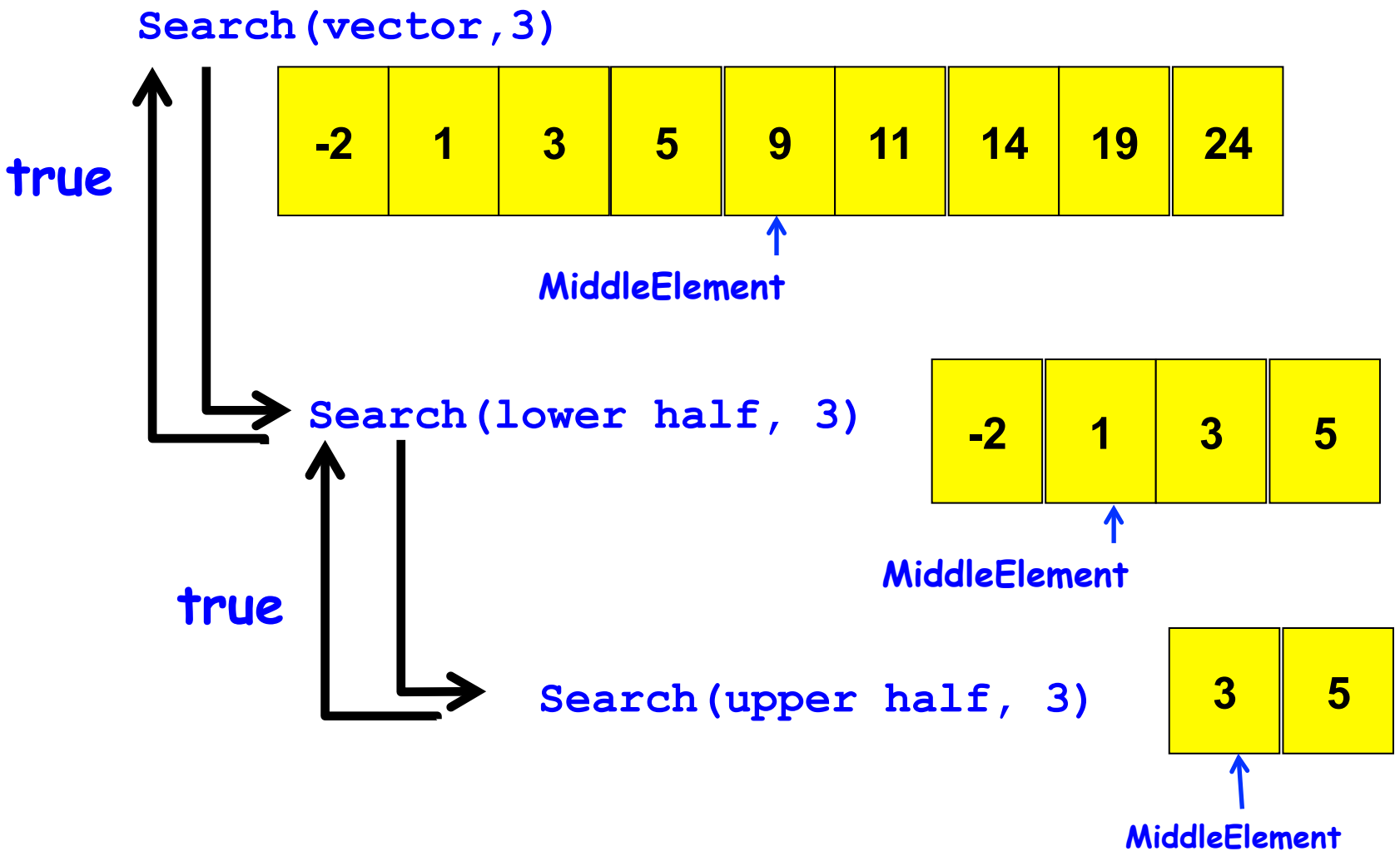
-2	1	3	5	9	11	14	19	24

```
bool Search (vector<T>, target)
        If vector<T> is empty
                return false
        Else If middle element == target
                return true
        Else If target > middle element
                return Search(upper half, target)
        Else
                return Search(lower half, target)
```

Is “18” in this vector?

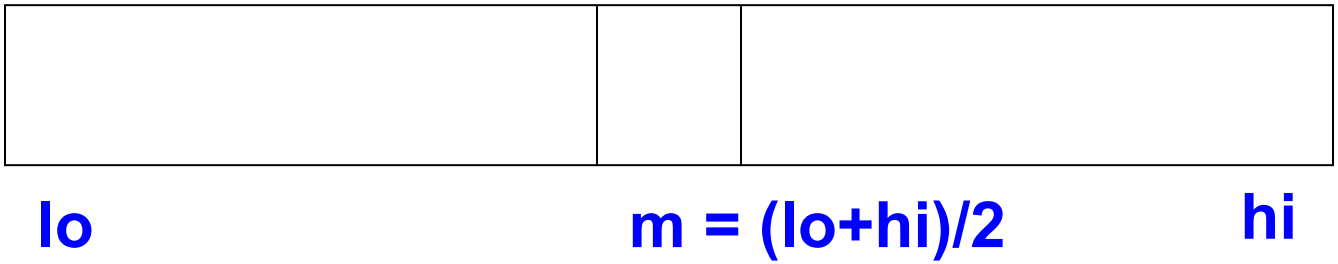


Is “3” in this vector?



Binary Search Implementation

Looking for `X`



If `hi - low < 0`, vector is empty,

so return `false`

If `a[m] == X`, we found `X`!,

so return `true`

If `X < a[m]` recursively search `a[lo..m-1]`

If `X > a[m]` recursively search `a[m+1..hi]`

Binary Search can be implemented recursively …

```
bool bSearch(vector<int> & a,
             int lo, int hi, int X)
{
   if (hi - lo < 0) return false;
   int m = (lo + hi) /2;
   if (X == a[m]) return true;  // found

   if (X < a[m])

      return bSearch(a, lo, m-1, X);

   else

      return bSearch(a, m+1, hi, X);
}
```

Final word: what we learned applies to many languages

- I've said many times that what we're learning is not just C++
  - By learning C++, we are learning concepts that are widely applicable

- Java: check out Recursion.java
- C#: check out Recursion.cs
- And many more: Python, PHP, Ruby, Objective-C, …