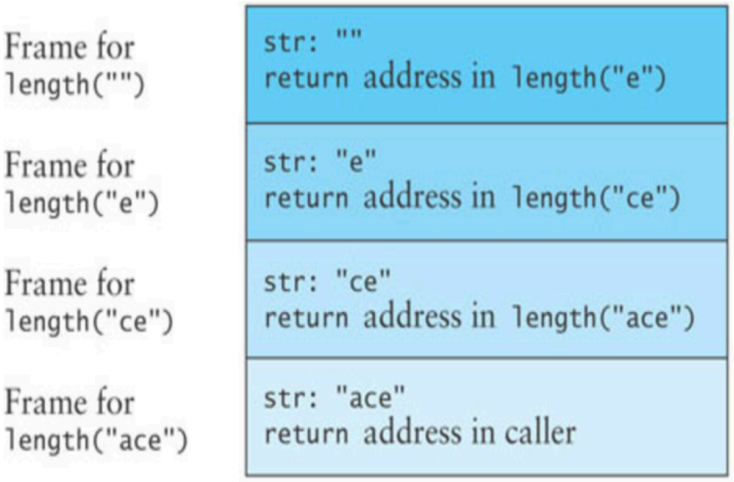
Recursion

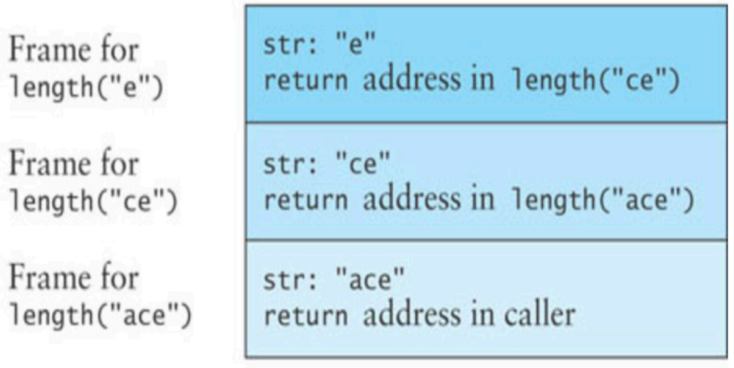
How does recursion work ?

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack

Runtime stack during invocation of length("ace")

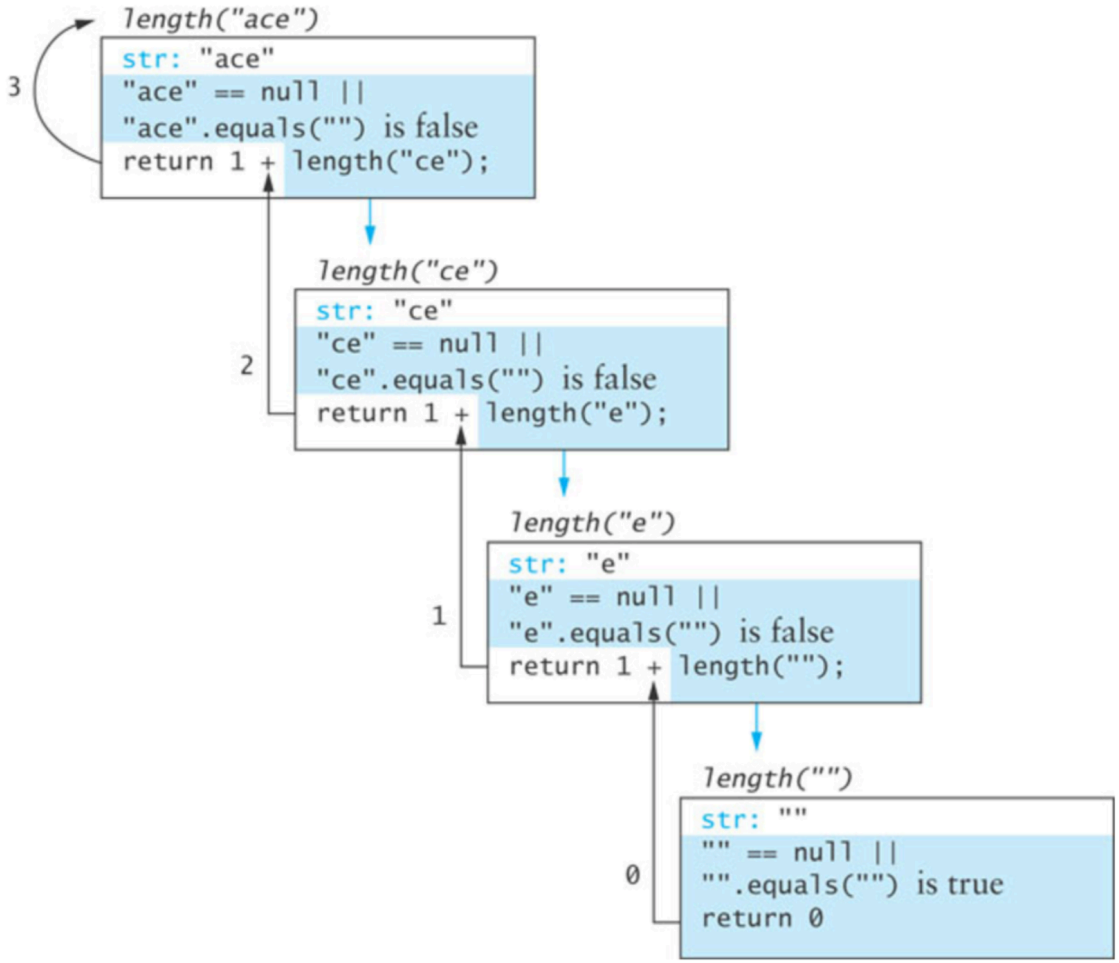


Run-time stack after all calls



Run-time stack after return from last call

Call invocations in length("ace")



Recursive definitions of Mathematical formulas

- Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms
- Examples include:
  - factorials
  - powers
  - greatest common divisors (gcd)

Factorial function

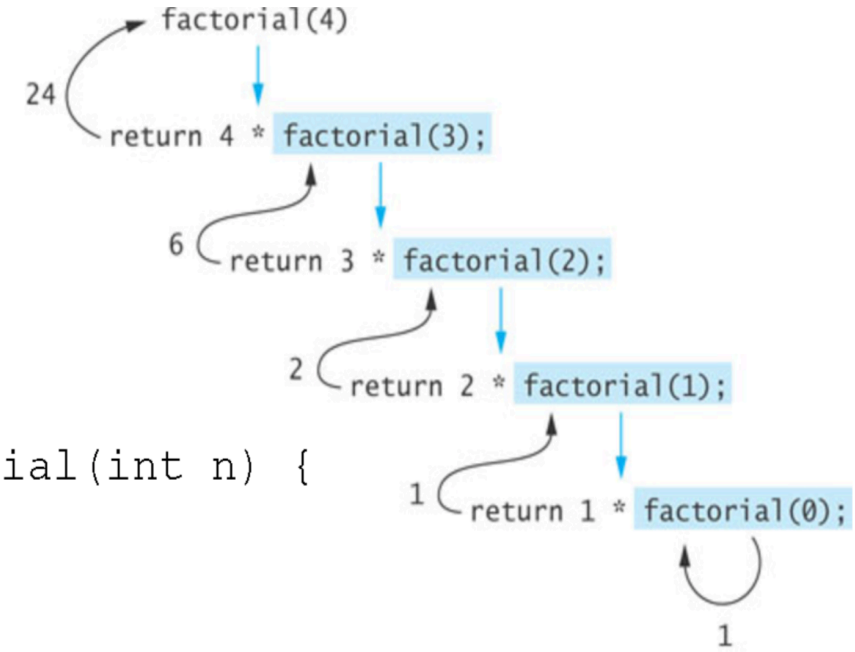
- The factorial of *n*, or *n*! is defined as follows:

  $0! = 1$

  $n! = n \times (n-1)!\ (n > 0)$

- The base case: *n* equal to 0
- The second formula is a recursive definition

factorial program




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

Avoid infinite recursion !!

- If you call method `factorial` with a negative argument, the recursion will not terminate because `n` will never equal `0`
- If a program does not terminate, it will eventually throw the `StackOverflowError` exception
- Make sure your recursive methods are constructed so that a stopping case is always reached
- In the `factorial` method, you could throw an `IllegalArgumentException` if `n` is negative

Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

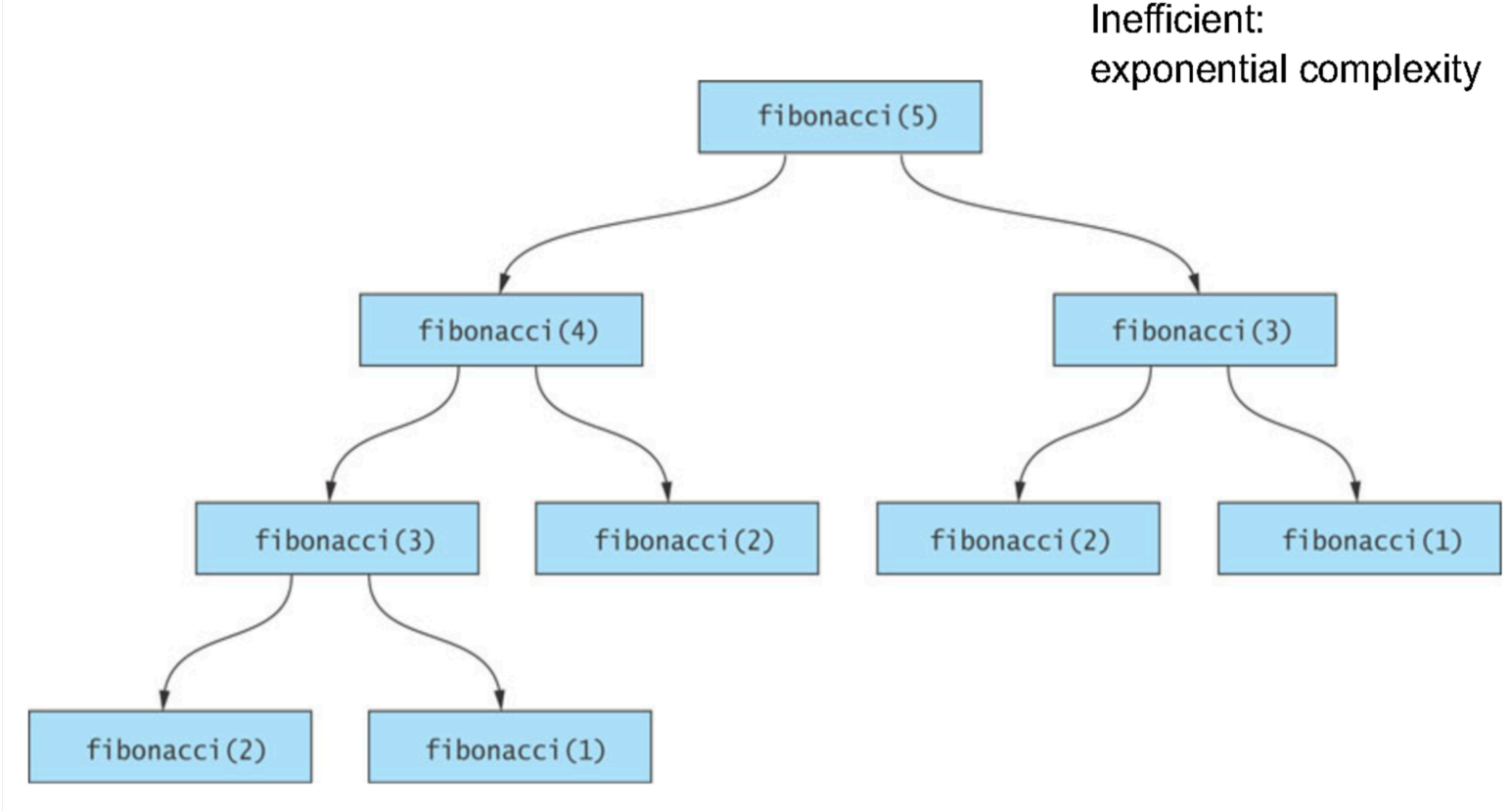
If we denote the ith Fibonacci number by F_i, $i = 0, 1, 2, \ldots$, then a recursive definition is given by:

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2,$$

with base cases $F_0 = 0$ and $F_1 = 1$.

```
public static int fib(int n) {
    // base cases
    if (n <= 1) {
        return n;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}
```

Recursive fibonacci



Recursion vs Iteration

- There are similarities between recursion and 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
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Iterative factorial function

```
public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i < n; i++) {
        result *= i;
    }
    return result;
}
```

Iterative Fibonacci function

```
public static int fibIter(int n) {
    if (n == 0) return n;
    int fib_nMinus2, fib_nMinus1 = 0;
    int fib_n = 1;
    for (int i=2; i <= n; i++) {
        fib_nMinus2 = fib_nMinus1;
        fib_nMinus1 = fib_n;
        fib_n = fib_nMinus1 + fib_nMinus2;
    }
    return fib_n;
}
```

Efficiency of recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
- 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
- The reduction in efficiency does not outweigh the advantage of readable code that is easy to debug

Towers of Hanoi

- The Towers of Hanoi is a puzzle made up of three vertical pegs and several disks that slide on the pegs
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one peg to another under the following rules:
  - We can move only one disk at a time
  - We cannot move a larger disk on top of a smaller one
- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant

TowersOfHanoi recursive function

```
public static void towersOfHanoi(int fromPeg, int
                                 toPeg, int nDisks) {
    if (nDisks == 1) {
        System.out.println(String.format("Move top
disk from peg %d to peg %d", fromPeg, toPeg));
        return;
    }
    int otherPeg = 6 - (fromPeg + toPeg);
    towersOfHanoi(fromPeg, otherPeg, nDisks - 1);
    towersOfHanoi(fromPeg, toPeg, 1);
    towersOfHanoi(otherPeg, toPeg, nDisks- 1);
}
```

Decision problems

- Determine a sequence of decisions that reaches a goal optimizes some measure or meets some constraint
- Backtracking search approach

-- exploring a sequence of possibilities until we reach the goal or reach a dead-end(no solution)

-- we backtrack one step and explore alternate path again.

- Recursion allows this to be done very easily and elegantly.

Navigating a maze

- Finding a path from origin to the destination of a maze. Possible directions to go : N,S,E,W

Backtrack search for a maze

- **Problem** : Is there a path from (0,0) to (N-1,N-1) ?

Subproblem – Is there a path from (x,y) to (N-1,N-1) where $0 \leq x,y \leq N-1$?

- **Recursive algorithm** :

Base case :

(a) If from (x,y) there is no valid neighbor to go to, no path

(b) If x = N-1 and y = N-1, found a path.

Recursion case:

Check if there is path from (x-1,y) or from (x+1, y), or from (x,y-1), or from (x,y+1) recursively.