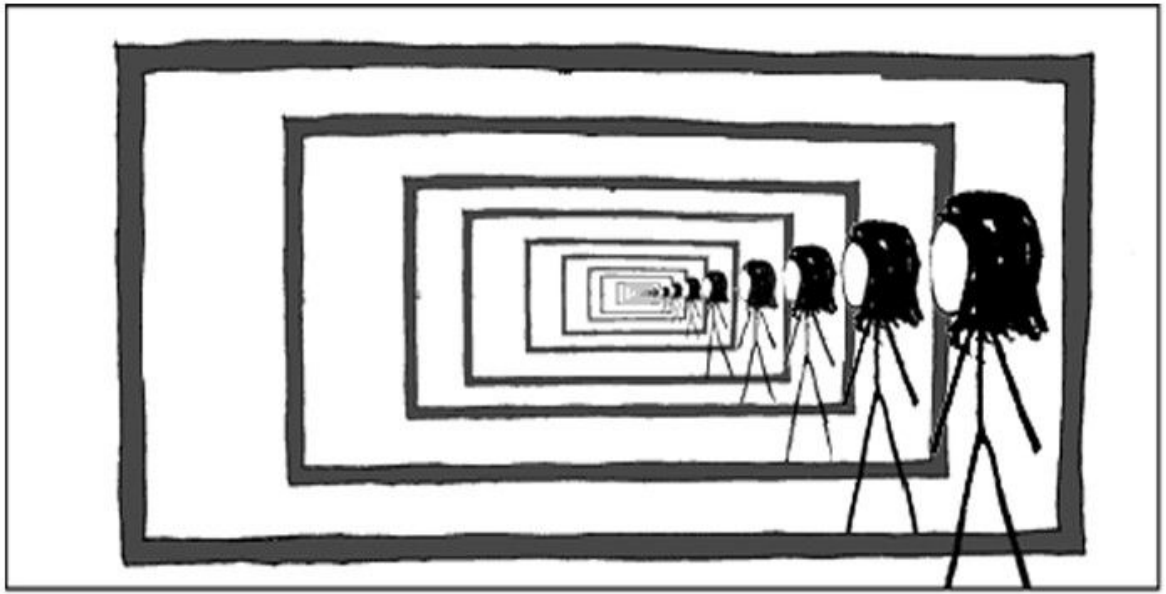
COSC 121: Computer Programming II



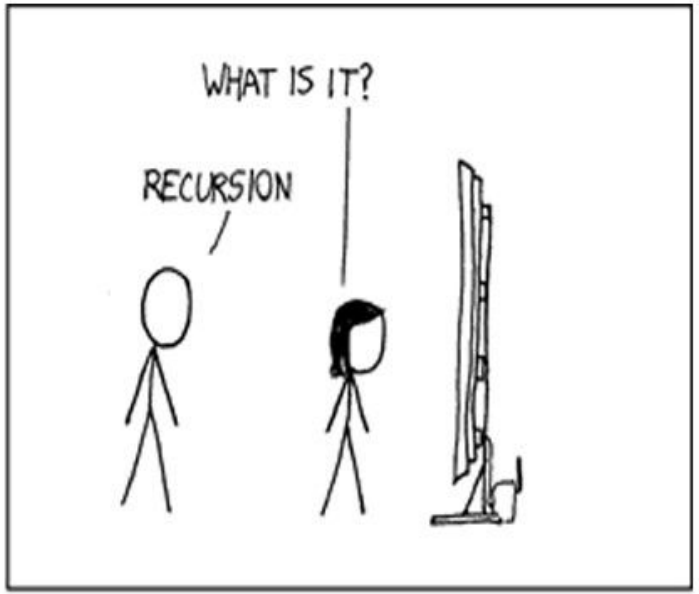


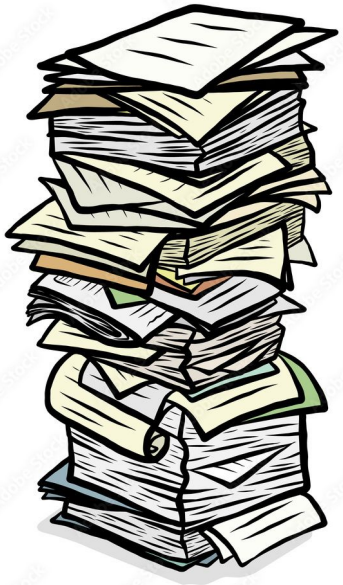
Image taken from linkedin.com

Today's Key Concepts

- Recursion is a technique where a function solves a problem by calling a smaller, simpler version of itself until a trivial base case is reached
- General structure:
  - Base case
  - Divide the problem up
  - Solve part of the problem
  - Call itself again (but with the smaller problem)
- Alternative approach to iteration
- Technique used in various data structures and algorithms
- Tail-recursion is a space-efficient version of recursion

Motivating Example: Exam Sorting

- After grading, exams are a mess but I need them sorted by last name. Let's say you are the TA coordinator and there are other TAs in the room who can help you. What should you do to get things sorted efficiently?



Motivating Example: Exam Sorting

- After grading, exams are a mess but I need them sorted by last name. Let's say you are the TA coordinator and there are other TAs in the room who can help you. What should you do to get things sorted efficiently?
  - Split up the stack into smaller stacks, give smaller stack to each TA to sort, then "merge" sorted ones together

  - Recursion makes use of the same idea, just applied to itself

Recursive Thinking

- Consider the following list of numbers:

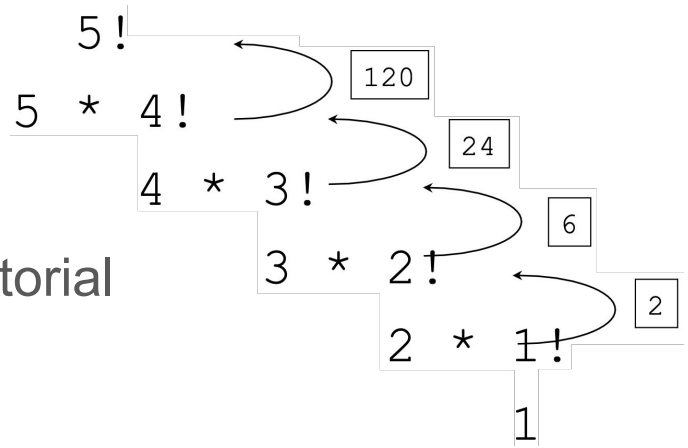
  24, 88, 40, 37
- How might we use recursion to define what a `List` is, following this syntax?
- Such a list can be defined as follows:

```
A List is a:    number
        or a:   number , List
```

- The concept of a List is used to define itself
  - Requires a stopping criterion called base case

Recall: Factorial from Math

- N! means, for any positive integer N, this expression is defined to be the product of all integers between 1 and N inclusive
- Examples:
  - 1! = 1
  - 2! = 1 x 2
  - N! = 1 x … x (N-1) x N
- A factorial is defined in terms of another factorial
  - e.g.: N! = N x (N-1)!
  - Eventually reaching the base case of 1!



Exercise

- Write a recursive definition for:
  5 x N, where N > 0
  - Base case?
  - Pattern?
  - Generalize common pattern?

Exercise

- Write a recursive definition for:
  5 x N, where N > 0
  - Base case: N = 1, then we have 5 x 1 = 5
  - Pattern?
  - Generalize common pattern?

Exercise

- Write a recursive definition for:
  5 x N, where N > 0
  - Base case: N = 1, then we have 5 x 1 = 5
  - Pattern:
    N = 2: 5 x 2 = 10
    N = 3: 5 x 3 = 15
    N = 4: 5 x 4 = 20
    N = 5: 5 x 5 = 25
  - Generalize common pattern?

Now what? 😅
Can we reframe based on N-1?

Exercise

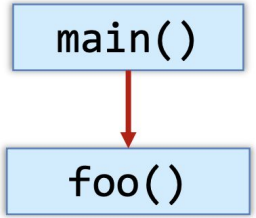
- Write a recursive definition for:
  5 x N, where N > 0
  - Base case: N = 1, then we have 5 x 1 = 5
  - Pattern:
    N = 2: 5 x 2 = 10 = 5 + (5 x 1)
    N = 3: 5 x 3 = 15 = 5 + (5 x 2)
    N = 4: 5 x 4 = 20 = 5 + (5 x 3)
    N = 5: 5 x 5 = 25 = 5 + (5 x 4)
  - Generalize common pattern?

Exercise

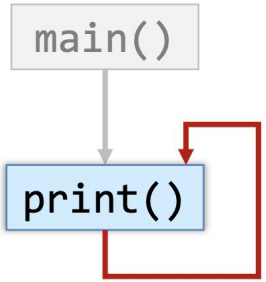
- Write a recursive definition for:
  5 x N, where N > 0
  - Base case: N = 1, then we have 5 x 1 = 5
  - Pattern:
    N = 2: 5 x 2 = 10 = 5 + (5 x 1)
    N = 3: 5 x 3 = 15 = 5 + (5 x 2)
    N = 4: 5 x 4 = 20 = 5 + (5 x 3)
    N = 5: 5 x 5 = 25 = 5 + (5 x 4)
  - Generalize common pattern: 5 x N = 5 + (5 x (N-1))

Recursion in Programming

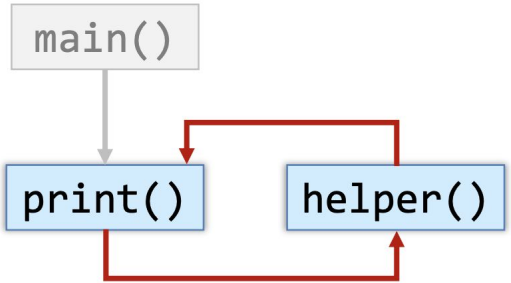
- Previously, you saw methods that call other methods:



- A recursive method is a method that calls itself:



calls itself directly



calls itself indirectly

Simple Recursion Code Examples

- Example:

```
public int question( int x, int y )
{
   if( x == y )
     return 0;
   else
     return question(x-1, y);
 }
```

- What would `question( 5, 3 )` return?

Tracing question(5, 3)

question(5, 3)

→ question(4, 3)

→ question(3, 3)

→ 0 ↩

↩

↩

```
public int question( int x, int y )
{
   if( x == y )
     return 0;
   else
     return question(x-1, y);
 }
```

Simple Recursion Code Examples

- Another example:

```
public int question( int x, int y )
{
   if( x == y )
     return 0;
   else
     return question(x-1, y) + 1;
 }
```

- What would `question( 5, 3 )` return?

Tracing question(5, 3)

question(5, 3)

→ question(4, 3) + 1

→ question(3, 3) + 1

→ 0 +1 ↩

→ 1 +1 ↩

→ 2 ↩

```
public int question( int x, int y )
{
   if( x == y )
     return 0;
   else
     return question(x-1, y) + 1;
 }
```

Programming Factorial

- Recall earlier recursive definition for: N! = N x (N-1)! where N > 0
  We found:
  - Base case of N = 1 is 1! = 1
  - General pattern for N! is N x (N-1)!
- Code:

```
public static int calcFactorial( int n )
{




}
```

Programming Factorial

- Recall earlier recursive definition for: N! = N x (N-1)! where N > 0
  We found:
  - Base case of N = 1 is 1! = 1
  - General pattern for N! is N x (N-1)!
- Code:

```
public static int calcFactorial( int n )
{

    return n * n-1;



}
```

Will this work?

Programming Factorial

- Recall earlier recursive definition for: N! = N x (N-1)! where N > 0
  We found:
  - Base case of N = 1 is 1! = 1
  - General pattern for N! is N x (N-1)!
- Code:

```
public static int calcFactorial( int n )
{

    return n * calcFactorial( n-1 );

}
```

What about this?

Programming Factorial

- Recall earlier recursive definition for: N! = N x (N-1)! where N > 0
  We found:
  - Base case of N = 1 is 1! = 1
  - General pattern for N! is N x (N-1)!
- Code:

```
public static int calcFactorial( int n )
{
    // base case
    if( n == 1 )
        return 1;



}
```

Programming Factorial

- Recall earlier recursive definition for: N! = N x (N-1)! where N > 0
  We found:
  - Base case of N = 1 is 1! = 1
  - General pattern for N! is N x (N-1)!
- Code:

```
public static int calcFactorial( int n )
{
    // base case
    if( n == 1 )
        return 1;
    else
        // general case
        return n * calcFactorial( n-1 )
}
```

Tracing calcFactorial(5)

```
public static int calcFactorial( int n )
{
    // base case
    if( n == 1 )
        return 1;
    else
        // general case
        return n * calcFactorial( n-1 );
}
```

calcFactorial(5)

→ 5 * calcFactorial(4)

→ 5 * (4 * calcFactorial(3))

→ 5 * (4 * (3 * calcFactorial(2)))

→ 5 * (4 * (3 * (2 * calcFactorial(1))))

→ 5 * (4 * (3 * (2 * 1))) ↩

→ 5 * (4 * (3 * 2)) ↩

→ 5 * (4 * 6) ↩

→ 5 * 24 ↩

→ 120 ↩

Programming 5 x N

- Recall earlier recursive definition for: 5 x N, where N > 0
  We found:
  - Base case of N = 1, with 5 x 1 = 5
  - General pattern for 5 x N to be 5 + (5 x (N-1))
- Code:

```
public static int fiveN( int n )
{

}
```

Programming 5 x N

- Recall earlier recursive definition for: 5 x N, where N > 0
  We found:
  - Base case of N = 1, with 5 x 1 = 5
  - General pattern for 5 x N to be 5 + (5 x (N-1))
- Code:

```
public static int fiveN( int n )
{
    int result = 0;

    return result;
}
```

Programming 5 x N

- Recall earlier recursive definition for: 5 x N, where N > 0
  We found:
  - Base case of N = 1, with 5 x 1 = 5
  - General pattern for 5 x N to be 5 + (5 x (N-1))
- Code:

```
public static int fiveN( int n )
{
    int result = 0;
    // base case


    // general case

    return result;
}
```

Programming 5 x N

- Recall earlier recursive definition for: 5 x N, where N > 0
  We found:
  - Base case of N = 1, with 5 x 1 = 5
  - General pattern for 5 x N to be 5 + (5 x (N-1))
- Code:

```
public static int fiveN( int n )
{
    int result = 0;
    // base case
    if( n == 1 )
        result = 5 * n;
    else
        // general case

    return result;
}
```

Programming 5 x N

- Recall earlier recursive definition for: 5 x N, where N > 0
  We found:
  - Base case of N = 1, with 5 x 1 = 5
  - General pattern for 5 x N to be 5 + (5 x (N-1))
- Code:

```
public static int fiveN( int n )
{
    int result = 0;
    // base case
    if( n == 1 )
        result = 5 * n;
    else
        // general case
        result = 5 + fiveN( n-1 );
    return result;
}
```

Tracing fiveN(4)

```
public static int fiveN( int n )
{
    int result = 0;
    // base case
    if( n == 1 )
        result = 5 * n;
    else
        // general case
        result = 5 + fiveN( n-1 );
    return result;
}
```

fiveN(4)

→ 5 + fiveN(3)

→ 5 + (5 + fiveN(2))

→ 5 + (5 + (5 + fiveN(1)))

→ 5 + (5 + (5 + (5 * 1)))

→ 5 + (5 + (5 + 5)) ↩

→ 5 + (5 + 10) ↩

→ 5 + 15 ↩

→ 20 ↩

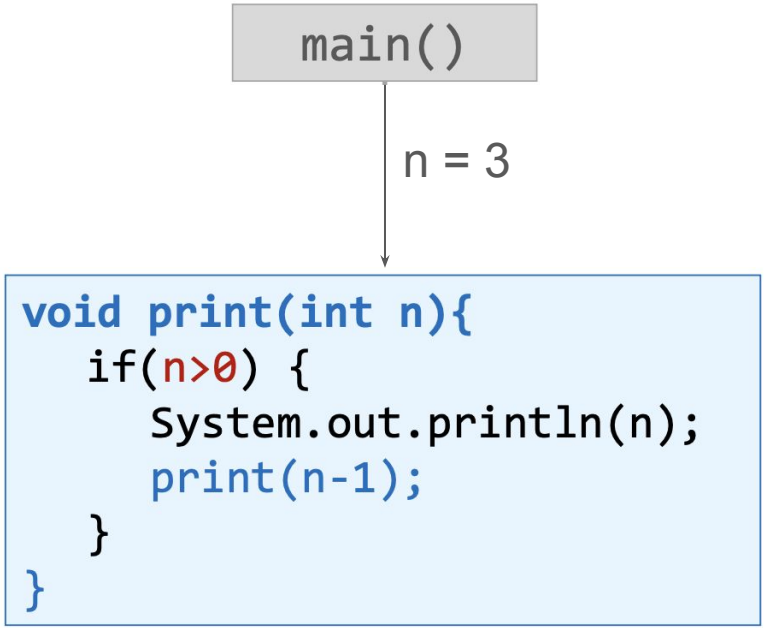
Printing Recursively

- This recursive method prints numbers from n down to 1

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

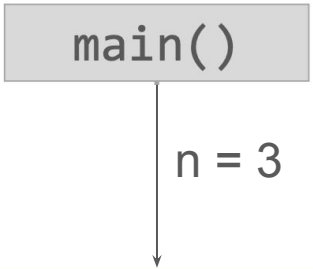
Printing Recursively

- This recursive method prints numbers from n down to 1

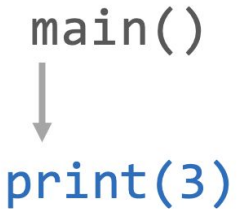


Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

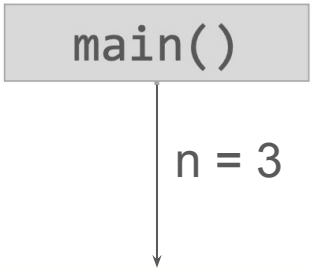


Output

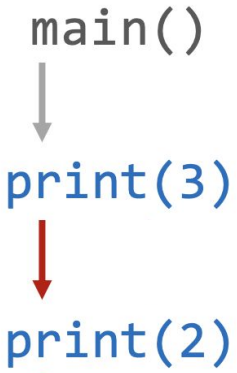
3

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

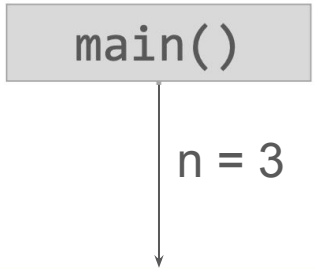


Output

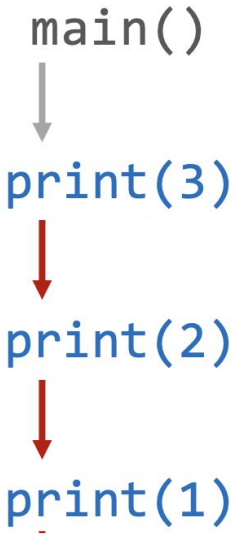
3 2

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

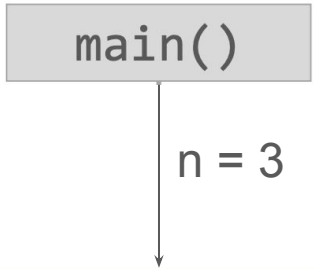


Output

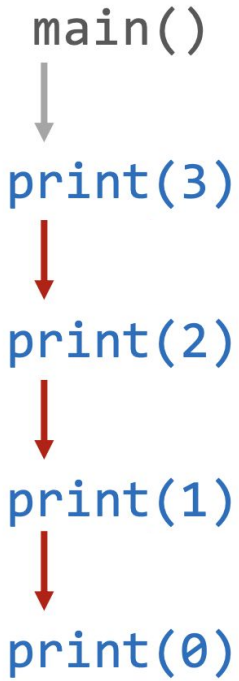
3 2 1

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

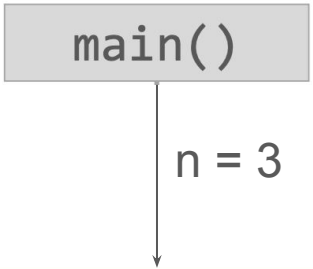


Output

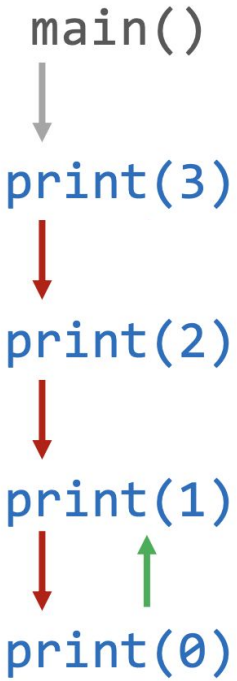
3 2 1

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

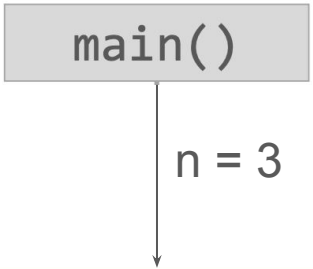


Output

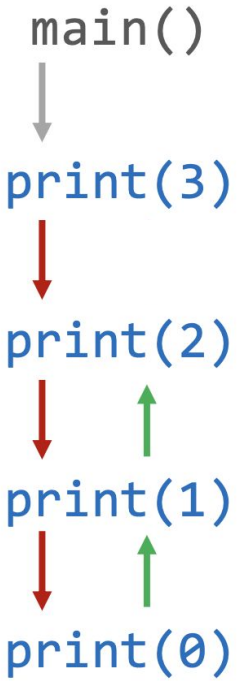
3 2 1

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

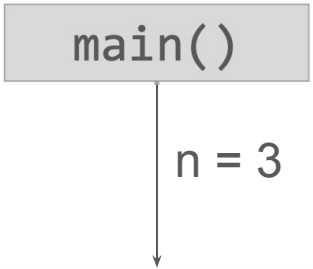


Output

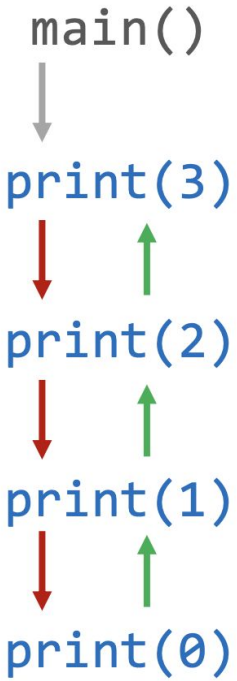
3 2 1

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

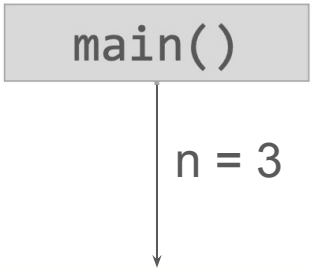


Output

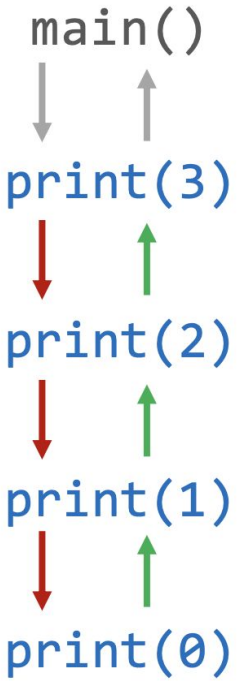
3 2 1

Printing Recursively

- This recursive method prints numbers from n down to 1




```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```



Output

3 2 1

iClicker Question

Which of these two code segments includes recursion?

A.

```
void foo(int a){
   if(a < 10)
     foo(a + 1);
}
```

B.

```
void foo(int a){
   if(a < 10)
      bar(a+1);
}

void bar(int b){
   b++;
}
```

iClicker Question

Which of these two code segments includes recursion?

A.

```
void foo(int a){
   bar(3);
}
void bar(int b){
   b++;
}
```

B.

```
void foo(int a){
   bar(3);
}
void bar(int b){
   foo(4);
}
```

iClicker Question

In the following method, what is the base case?

```
static int xMethod(int n) {
  if (n == 1)
    return 1;
  else
    return n + xMethod(n - 1);
}
```

A. n is 1
B. n is not 1
C. n is greater than 1
D. n is less than 1
E. no base case

Recursion vs. Iteration

- How to write a non-recursive version of print()?

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

Recursion vs. Iteration

- How to write a non-recursive version of print()?

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

```
void print(int n){
   for(int i = n; i > 0; i--)
      System.out.println(i);
}
```

- Both versions have a base case and general pattern

Recursion vs. Iteration

- How to write a non-recursive version of print()?

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

```
void print(int n){
   for(int i = n; i > 0; i--)
      System.out.println(i);
}
```

- Both versions have a base case and general pattern
- Any problem that can be solved with recursion can also be solved with iteration and vice versa
- Use recursion when there is a natural subproblem decomposition

Factorial: Recursive vs. Iterative Versions

- How to write a non-recursive version of factorial?

```
public static int calcFactorial( int n )
{
    // base case
    if( n == 1 )
        return 1;
    else
        // general case
        return n * calcFactorial( n-1 );
}
```

Factorial: Recursive vs. Iterative Versions

- How to write a non-recursive version of factorial?

```
public static int calcFactorial( int n )
{
    // base case
    if( n == 1 )
        return 1;
    else
        // general case
        return n * calcFactorial( n-1 );
}
```

```
public static int factorialLoop( int n )
{
    int result = 1;
    for( int i = n; i >= 1; i-- )
        result = result * i;
    return result;
}
```

- Use idea of accumulating the temporary result into a variable to reduce space

```
public static int factorialTail( int n ) {    return tailHelper( n, 1 );    }
private static int tailHelper( int n, int acc )
{
    if( n == 1 )
        return 1;
    else
        return tailHelper( n-1, acc*n );
}
```

this is called tail-recursion

iClicker Question

What is the return value for xMethod(4)?

```
static int xMethod(int n) {
  if (n == 1)
    return 1;
  else
    return n + xMethod(n - 1);
}
```

A. 9
B. 10
C. 11
D. 12
E. Something else

iClicker Question

What is the output if we call print(2) ?

```
void print(int n){
   if(n <= 5) {
      System.out.print(n);
      print(n+1);
   }
}
```

A. 543210
B. 012345
C. 2345
D. 234
E. 2

When to Use Recursion?

- When the problem has natural subproblems and problems that are difficult to solve using simple loops (linear)
  - Data structures: linked lists, trees, graphs, any sort of nested structures
  - Algorithms: merge sort, quicksort, binary search, tree and graph traversal methods, backtracking algorithms, dynamic programming