Priority Queues
Heaps

cs206

Oct 27

Priority Queue

- A queue that maintains order of elements according to some priority
  - Removal order, not general order
    - the rest may or may not be sorted

Complexity Analysis

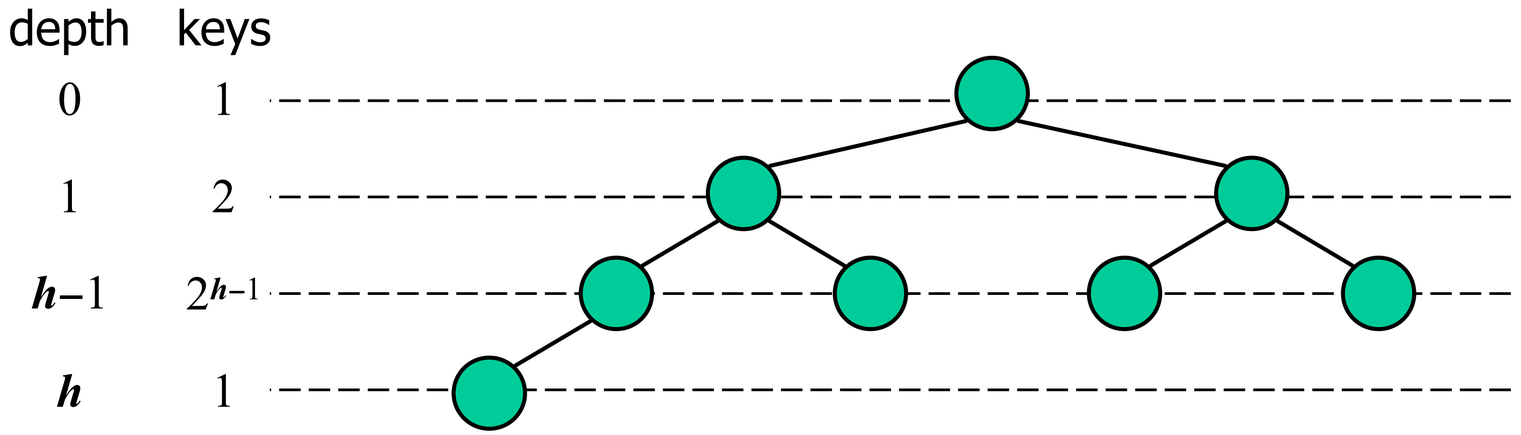
	Unordered	Ordered (using SL) (SL is hypothetical "sortedArray" class)
offer	O(1)	O(n)
peek	O(n)	O(1)
poll	O(n)	O(1)
Total time to offer/poll n items		

Binary Heap

- A heap is a "binary tree" storing keys at its nodes and satisfying:
  - heap-order: for every internal node v other than root, $key(v) \geq key(parent(v))$
  - "complete binary tree": let h be the height of the heap
    - Heap is filled from top down and within a level from left to right.
      - at depth h, the leaf nodes are in the leftmost positions
      - last node of a heap is the rightmost node of max depth

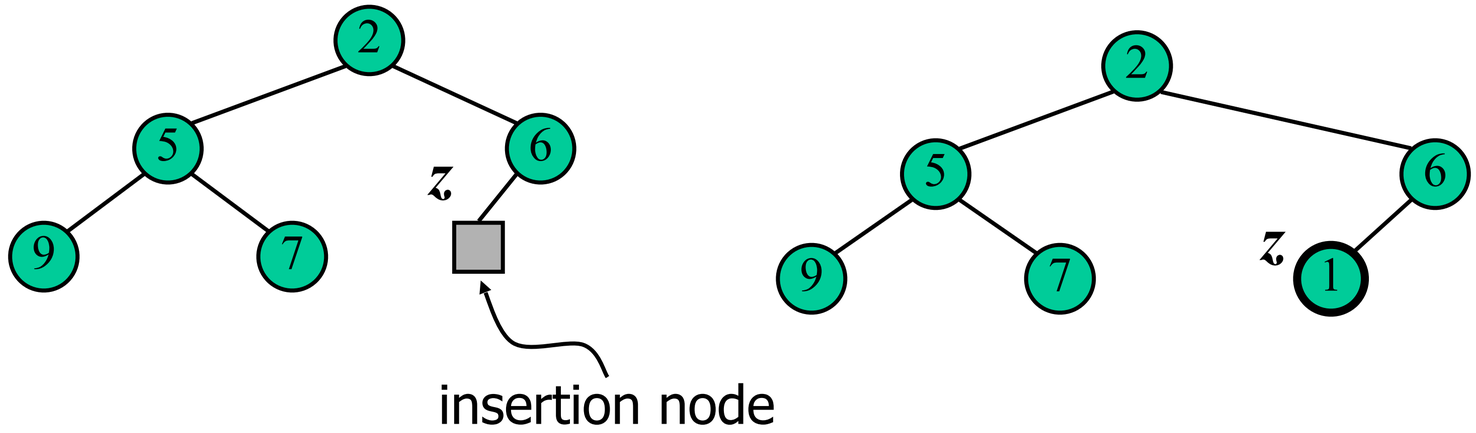
Height of a Heap

- A binary heap storing n keys has a height of $O(\log_2 n)$



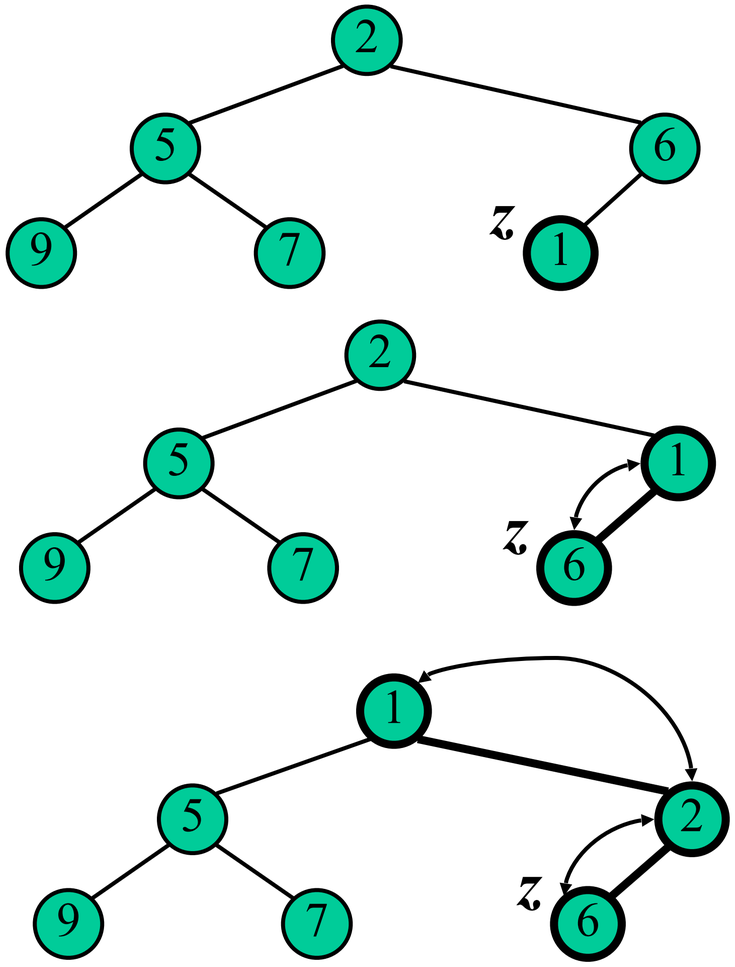
Insertion into a Heap

- Insert as new last node
- Need to restore heap order



Upheap

- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root
- $O(logn)$



Priority Queue using Heaps

startup

```
public class PriorityQHeap<K extends Comparable<K>, V> implements PriorityQInterface<K,V>
{

/** The default size of the heap.  This corresponds to a max depth or 10. */
    private static final int CAPACITY = 1032;
    /** The array that holds the heap. */
    private Pair<K,V>[] backArray;
    /** The number of items actually in he heap. */
    private int size;
    /** The way in which the heap is ordered */
    final private Ordering order;

   public PriorityQHeap() {
        this(Ordering.MIN, CAPACITY);
    }

   public PriorityQHeap(Ordering order, int capacity) {
        this.order=order;
        backArray = new Pair[capacity];
    }
```

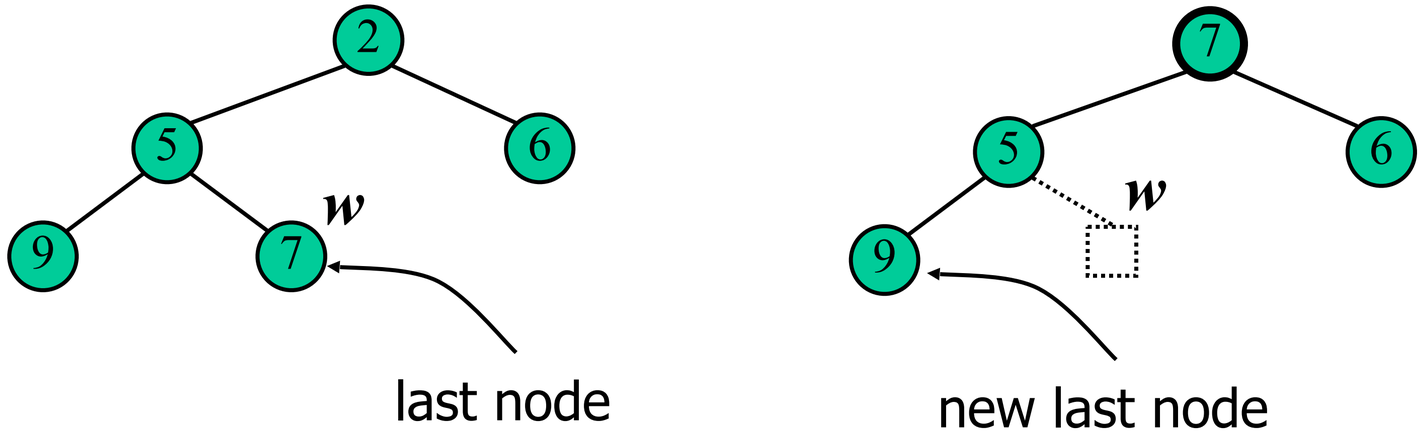
Heap Insertion

Priority Queue offer method

```
public boolean offer(K key, V value) {
    if (size>=(backArray.length-1))
        return false; // no space in the array
    // put in at end
    int loc = size++;
    backArray[loc] = new Pair<K,V>(key, value);
    // up heap
    int upp = (loc-1)/2;
    while (loc!=0) {
        if (0 > backArray[loc].doCompare(backArray[upp])) {
            // swap and climb
            Pair<K,V> tmp = backArray[upp];
            backArray[upp] = backArray[loc];
            backArray[loc] = tmp;
            loc = upp;
            upp = (loc-1)/2;
            }
        else {
            break;
        }
    }
    return true;
    }
```

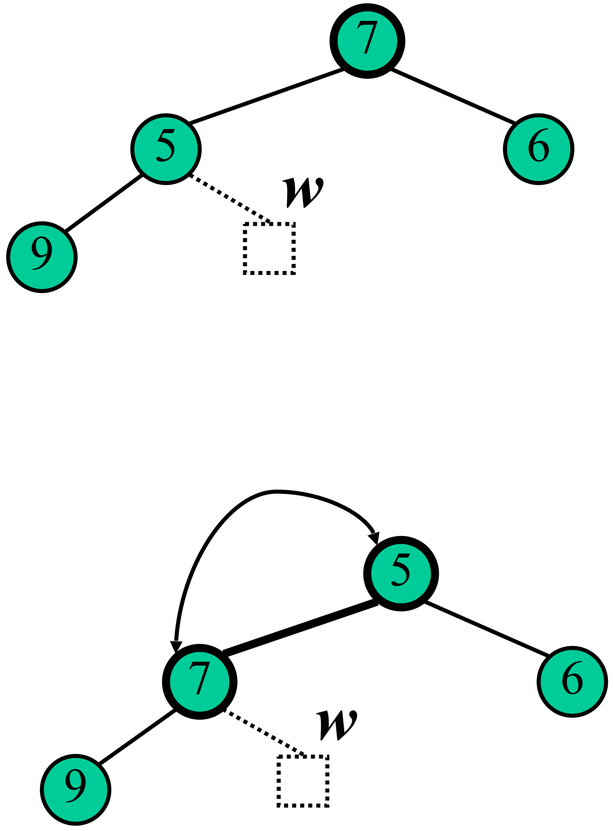
Poll

- Removing the root of the heap
  - Replace root with last node
  - Remove last node w
  - Restore heap order



Downheap

- Restore heap order
  - swap downwards
  - swap with smaller child
  - stop when finding larger children
  - or reach a leaf
- $O(logn)$



Peek and Poll

```
@Override
    public V poll() {
        if (isEmpty())
            return null;
        Entry<K,V> tmp = backArray[0];
        removeTop();
        return tmp.theV;
    }

    @Override
    public V peek() {
        if (isEmpty())
            return null;
        return backArray[0].theV;
    }
```

Remove head item from Heap

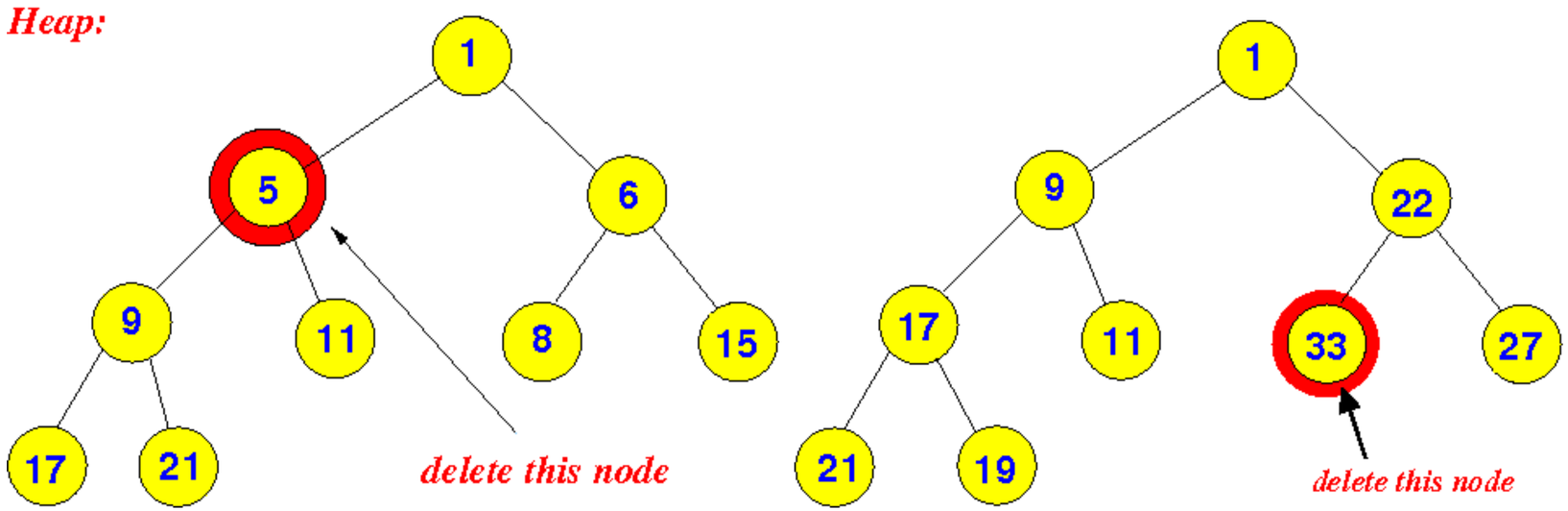
```
private void removeTop()
{
    backArray[0] = backArray[size-1];
    backArray[size-1]=null;
    size--;
    int upp=0;
    while (true)
    {
        int dwn;
        int dwn1 = upp*2+1;
        if (dwn1>size) break;
        int dwn2 = upp*2+2;
        if (dwn2>size) {  dwn=dwn1;
        } else {
            int cmp = backArray[dwn1].compareTo(backArray[dwn2]);
            if (cmp<=0)  dwn=dwn1;
            else dwn=dwn2;
        }
        if (0 > backArray[dwn].compareTo(backArray[upp]))
        {
            Pair<K,V> tmp = backArray[dwn];
            backArray[dwn] = backArray[upp];
            backArray[upp] = tmp;
            upp=dwn;            }
        else {  break;            } } }
```

Complexity Analysis

	Unordered	Ordered (using SAL)	Heap Based
offer	O(1)	O(n)	
peek	O(n)	O(1)	
poll	O(n)	O(1)	
Total time to offer/ poll n items			

General Removal

- swap with last node
- delete last node
- may need to upheap or downheap



Recursion

Any method that calls itself, either directly or indirectly

Idea, take a problem,
break that problem down into a slightly simpler problem,
ask yourself to solve that slightly simpler problem,
repeat

Factorials

- Recursive definition: $f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$

- Java method

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

Recursive Method

- Base case(s):
  - no recursive calls are performed
  - every chain of recursive calls must reach a base case eventually
- Recursive calls:
  - Calls to the same method in a way that progress is made towards a base case

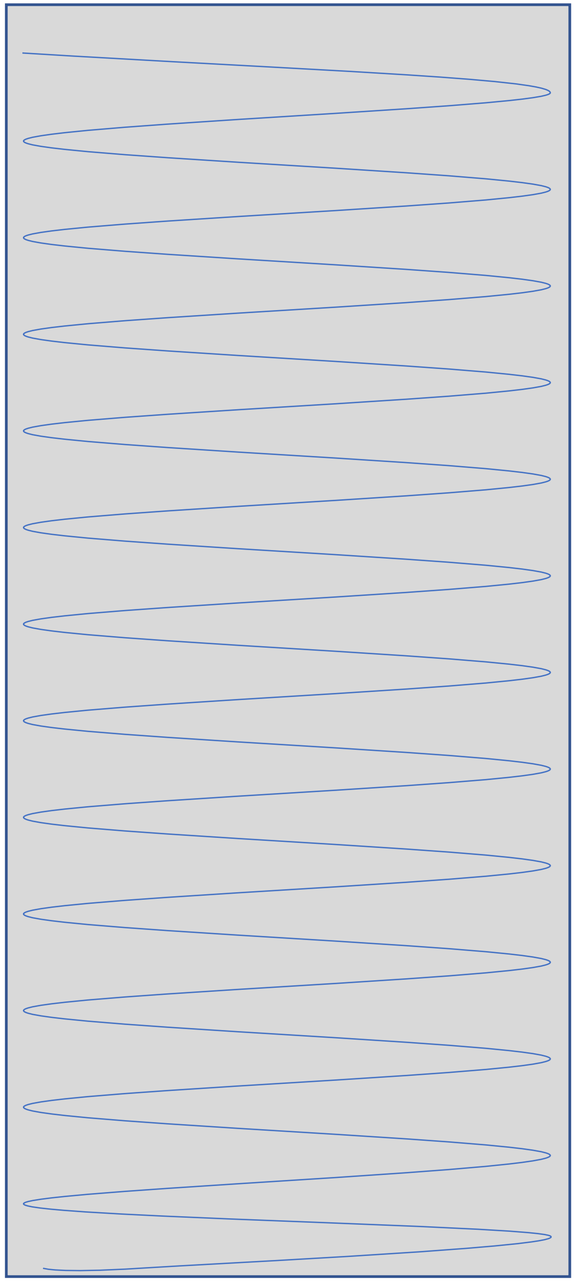
Compiled Code

```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

```
1.   int factorial(int n) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n *
     factorial(n-1);
6.       return F;
7.     }
8.   }
```

Executing Function

Call Stack



Compiled Code

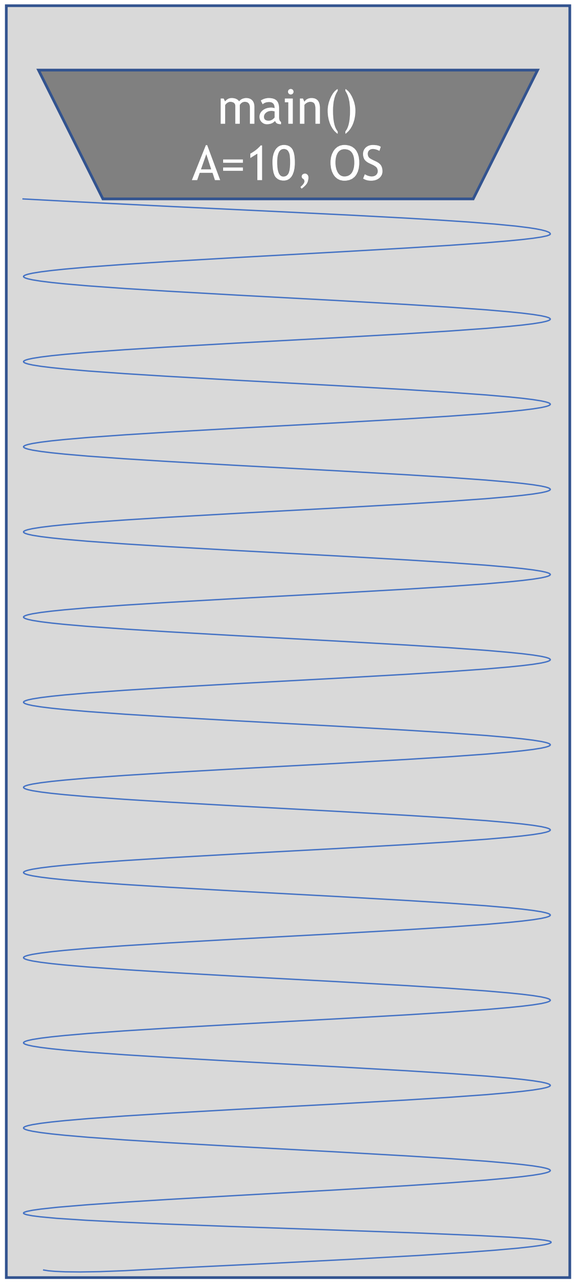
```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

```
1.   int factorial(int n) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n *
     factorial(n-1);
6.       return F;
7.     }
8.   }
```

Executing Function

```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

Call Stack



Compiled Code

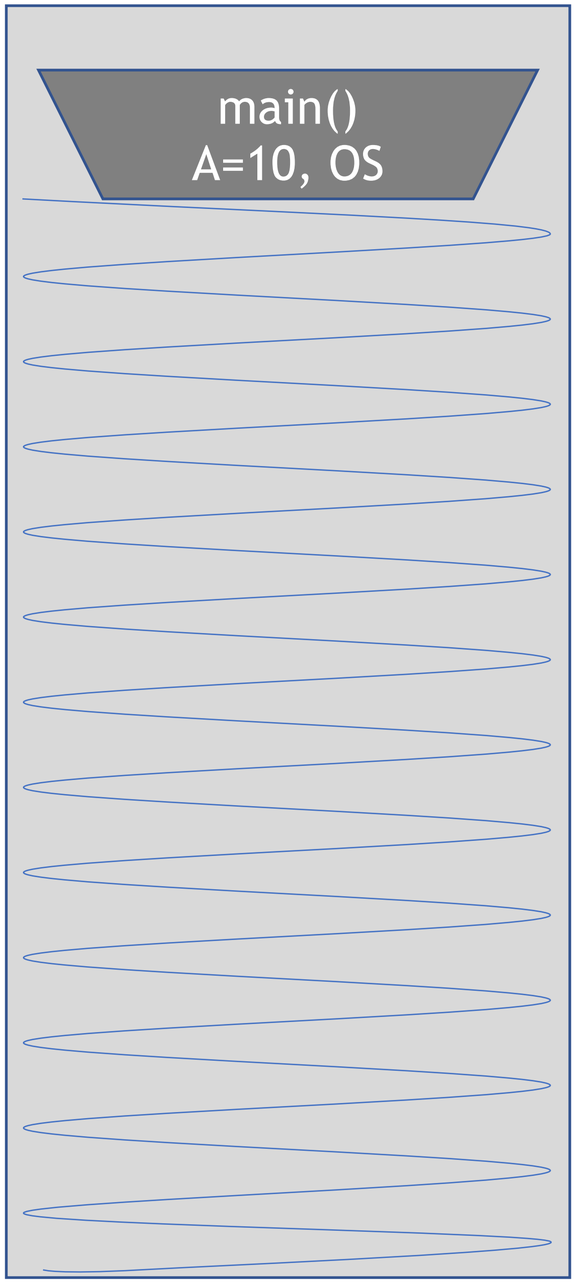
```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

```
1.   int factorial(int n) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n *
     factorial(n-1);
6.       return F;
7.     }
8.   }
```

Executing Function

```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

Call Stack



Compiled Code

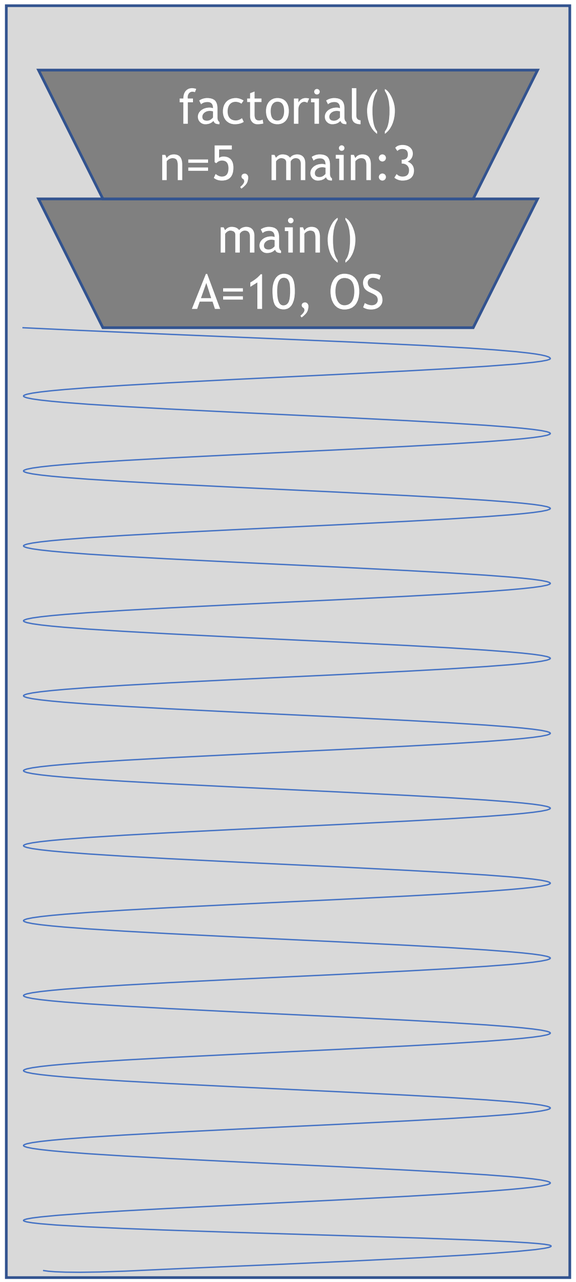
```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

```
1.   int factorial(int n) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n *
     factorial(n-1);
6.       return F;
7.     }
8.   }
```

Executing Function

```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

Call Stack



Compiled Code

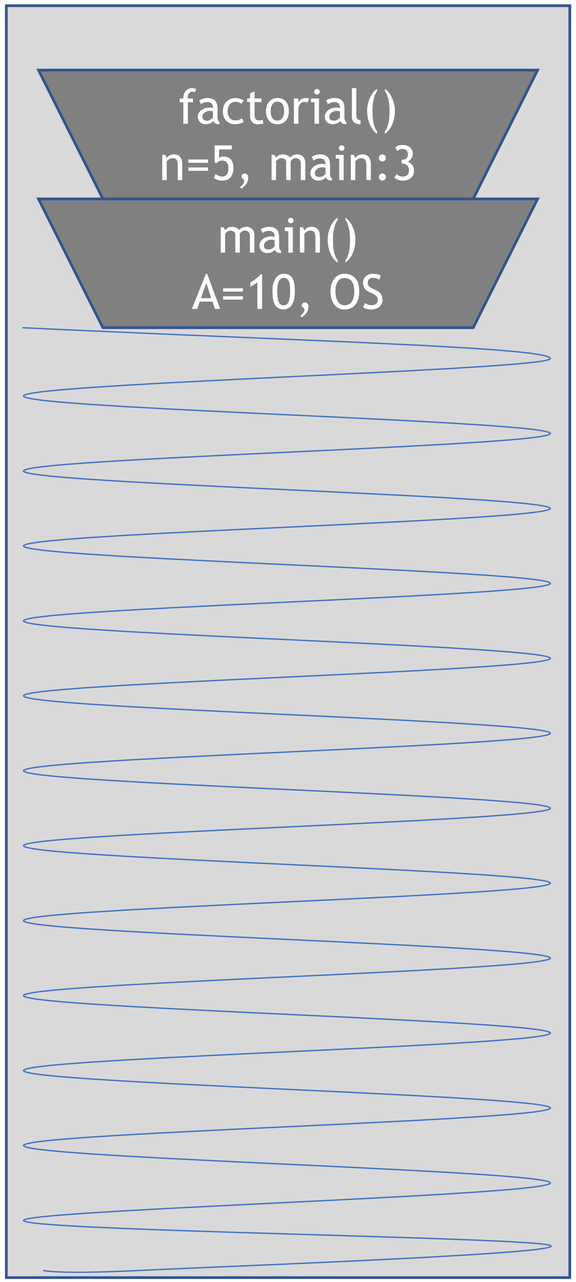
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=5) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

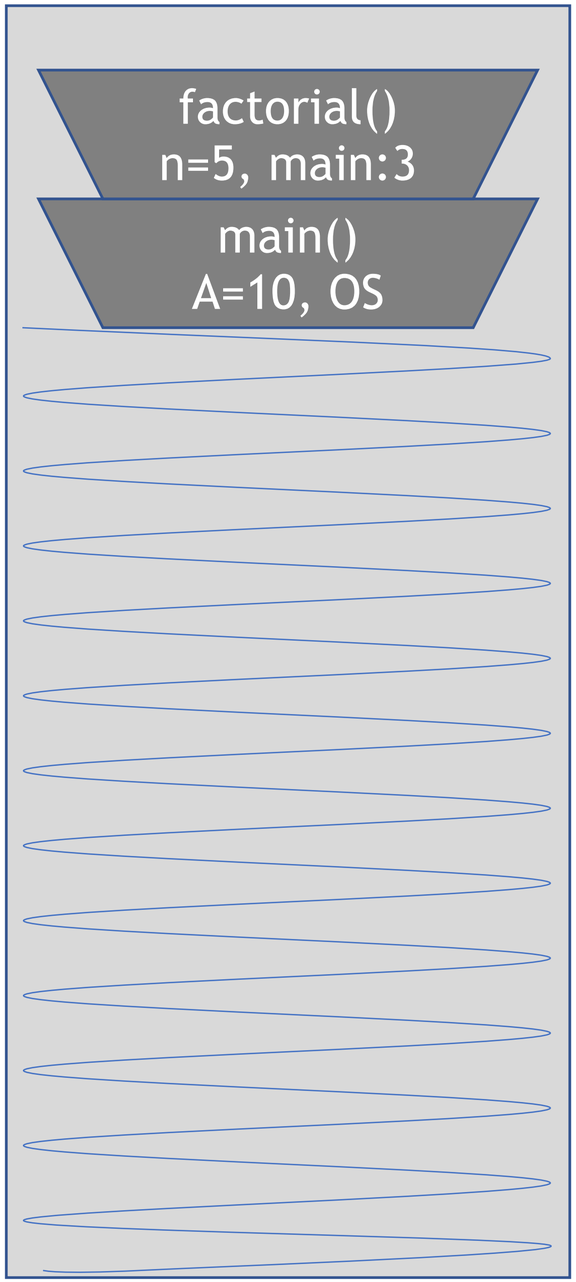
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=5) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

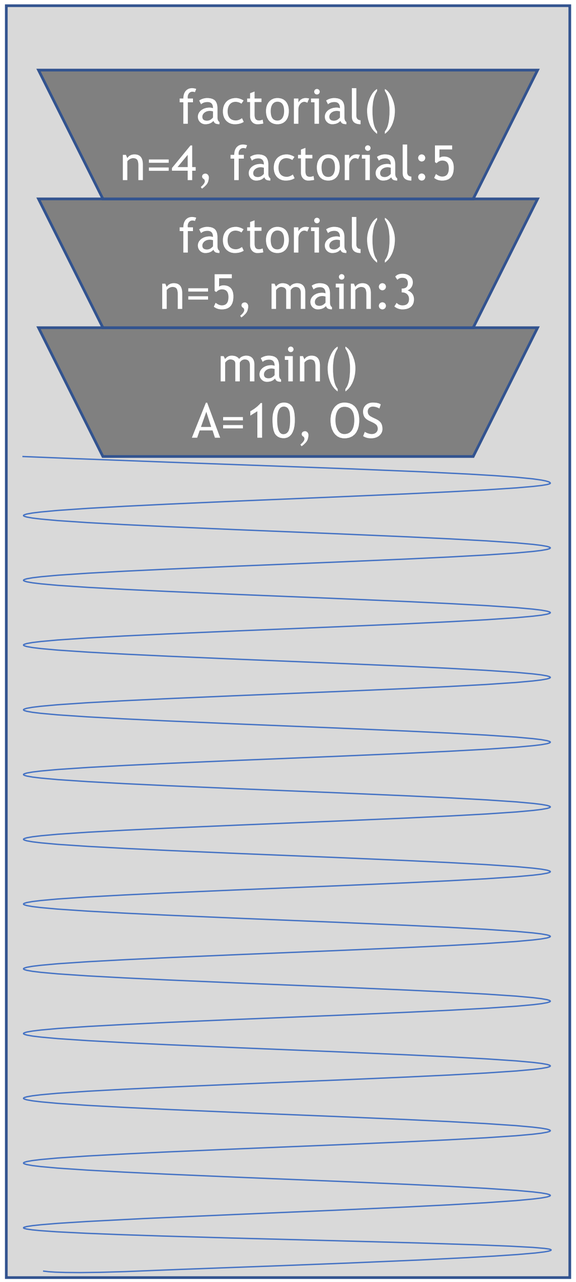
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=5) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

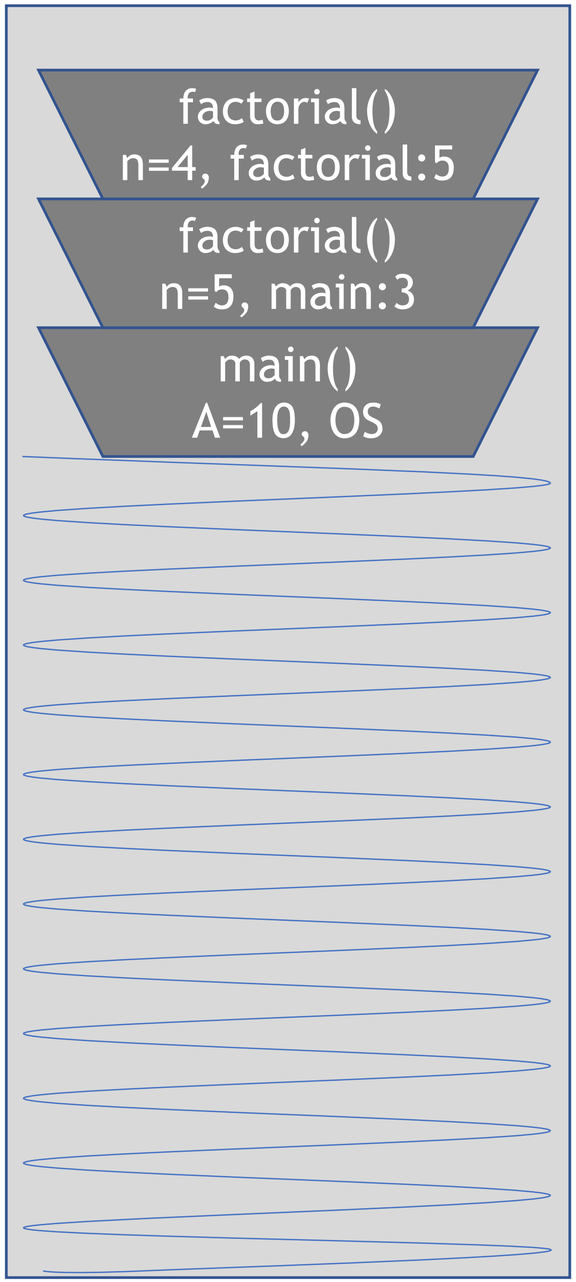
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=4) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

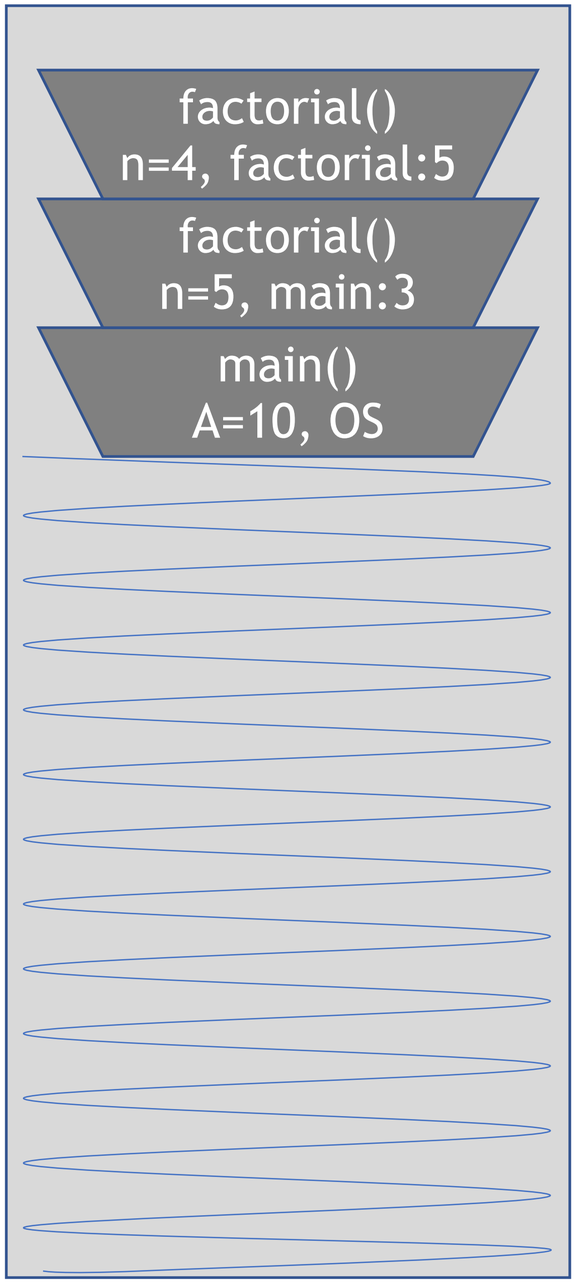
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=4) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

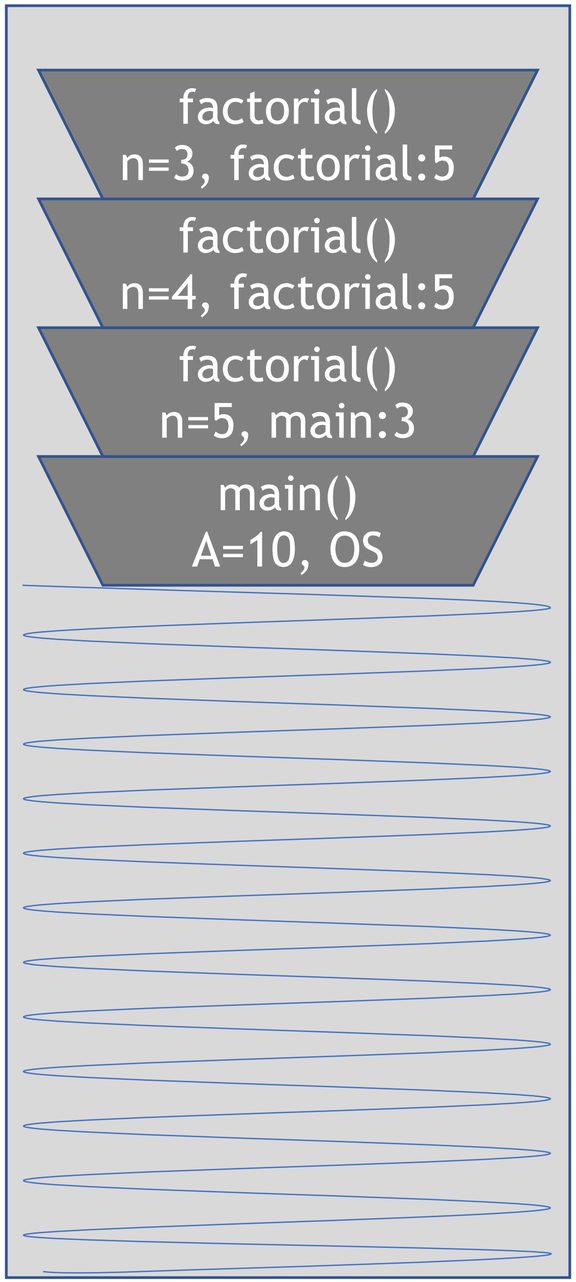
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=4) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

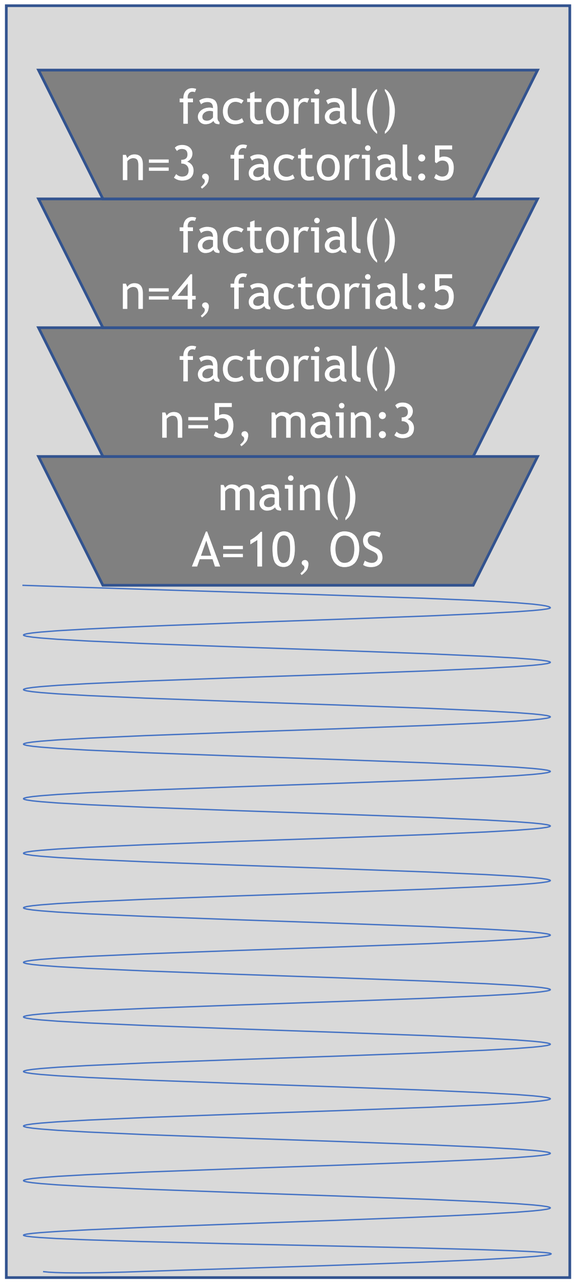
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=3) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

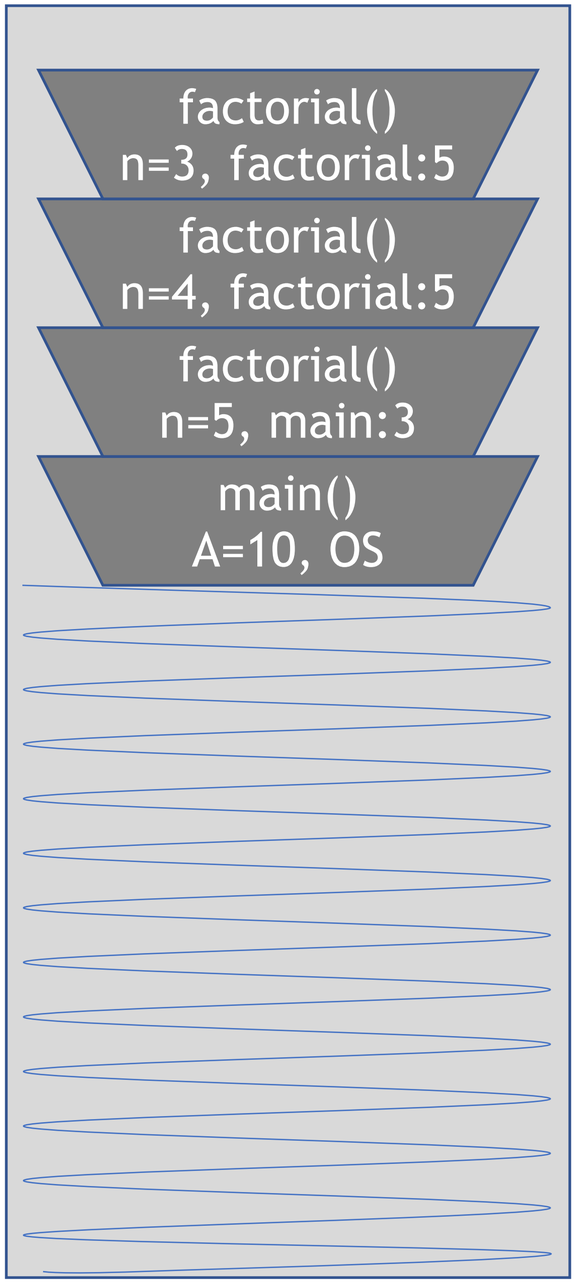
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=3) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

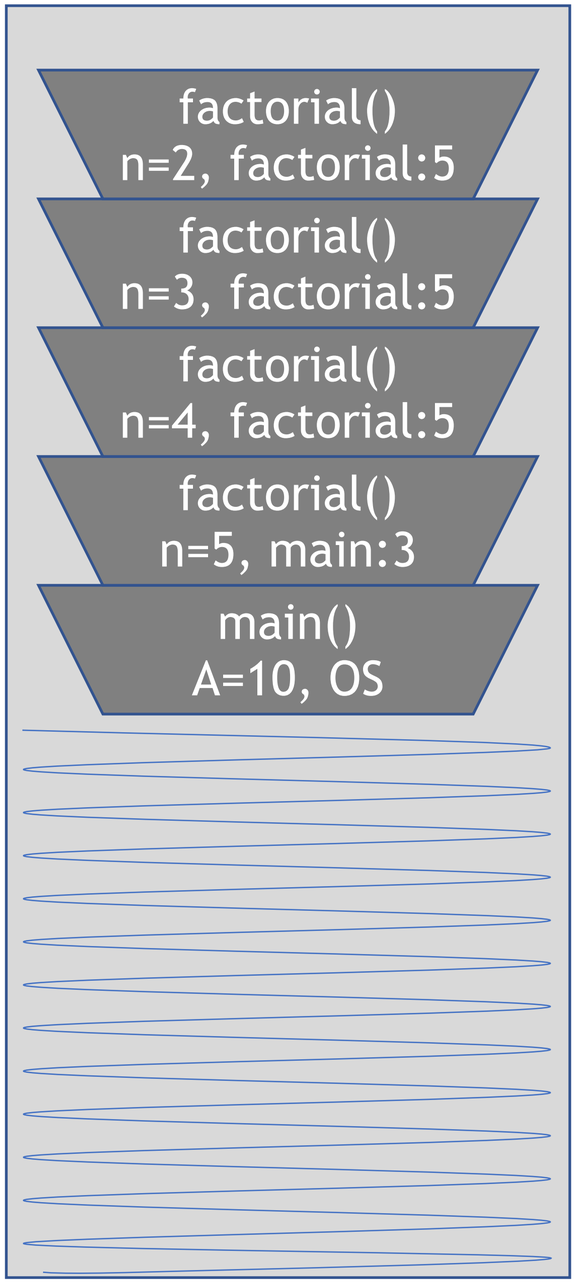
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=3) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

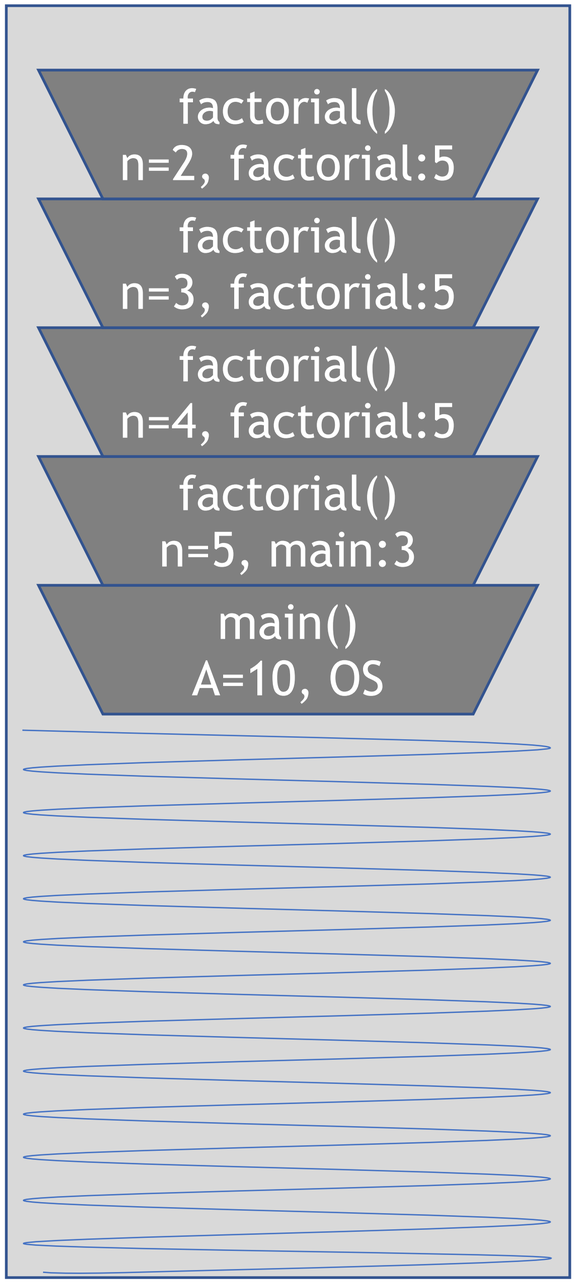
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=2) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

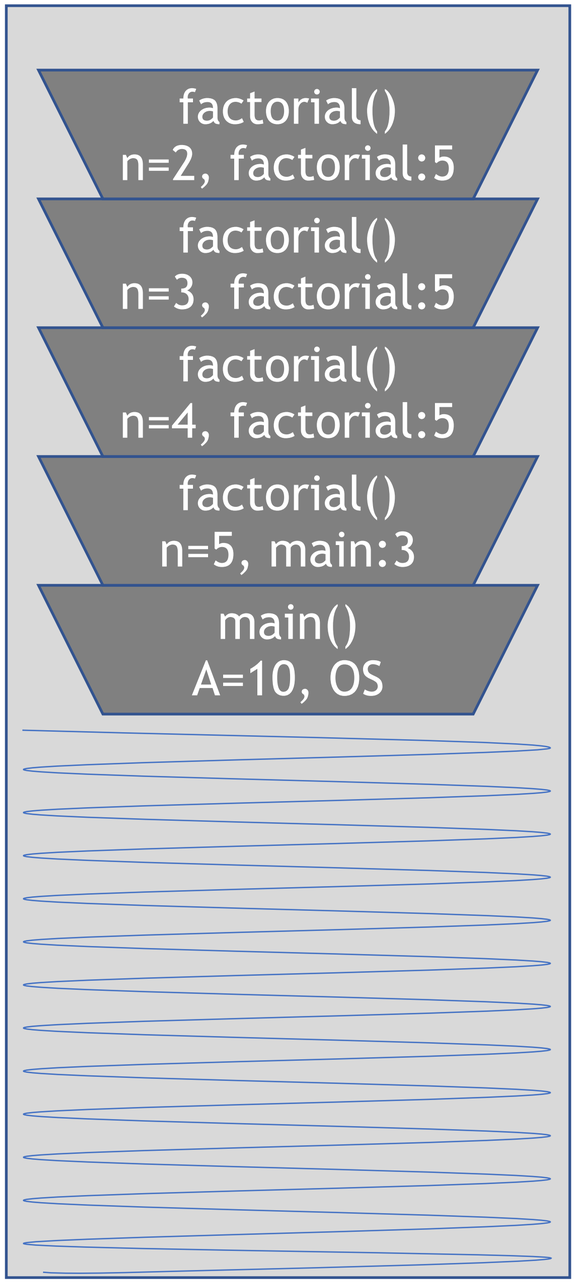
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=2) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



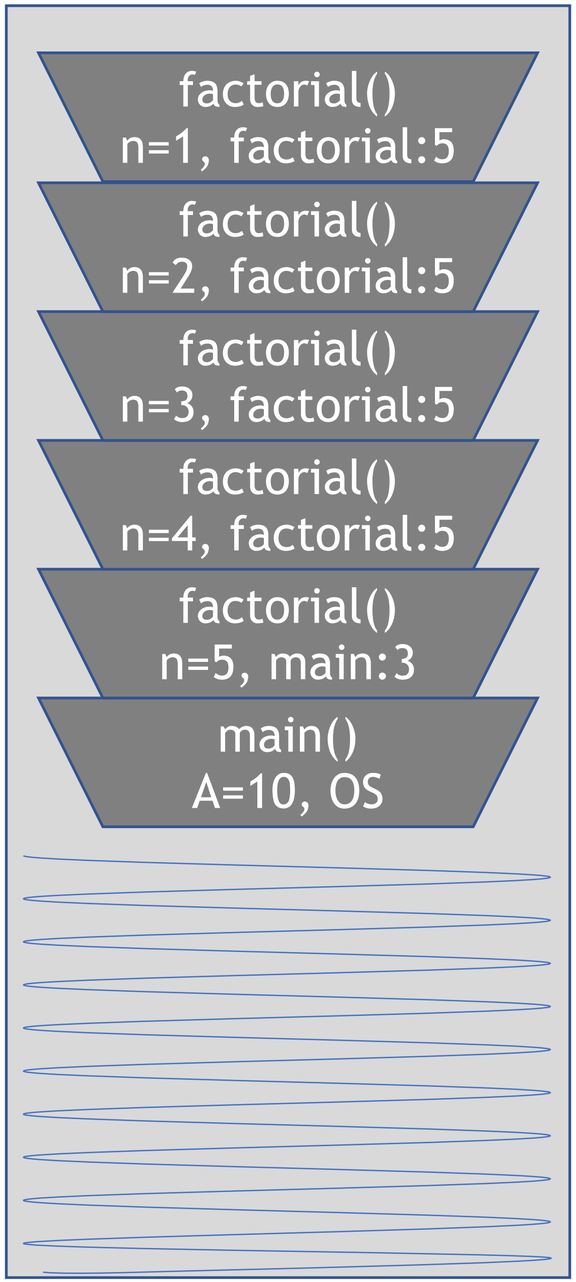
Compiled Code

```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=2) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```



Compiled Code

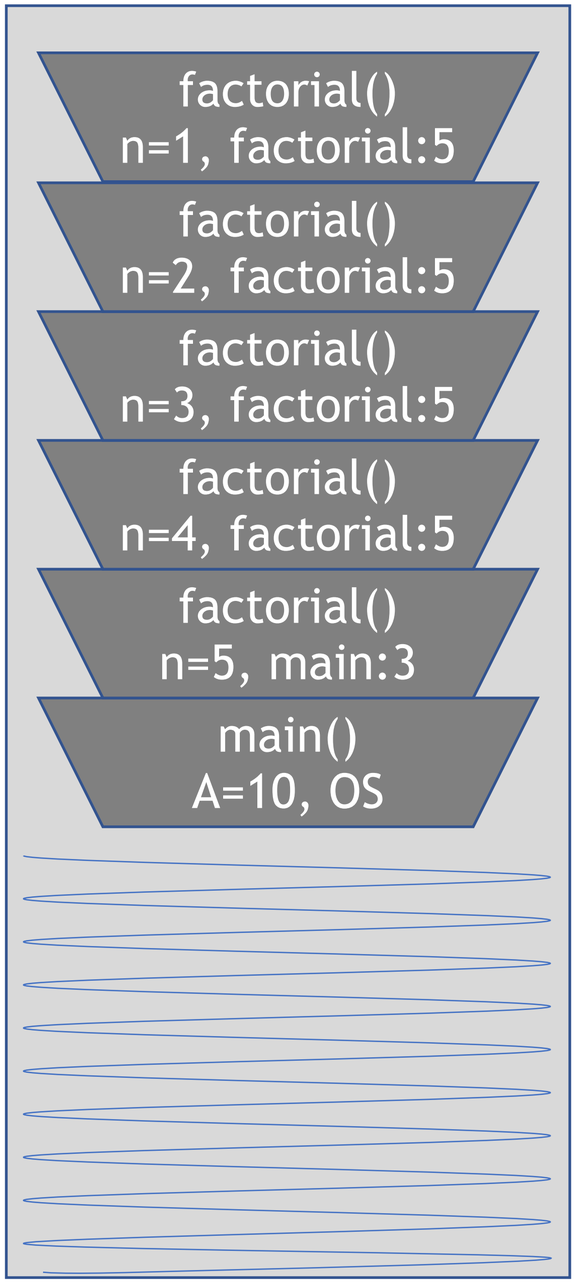
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=1) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

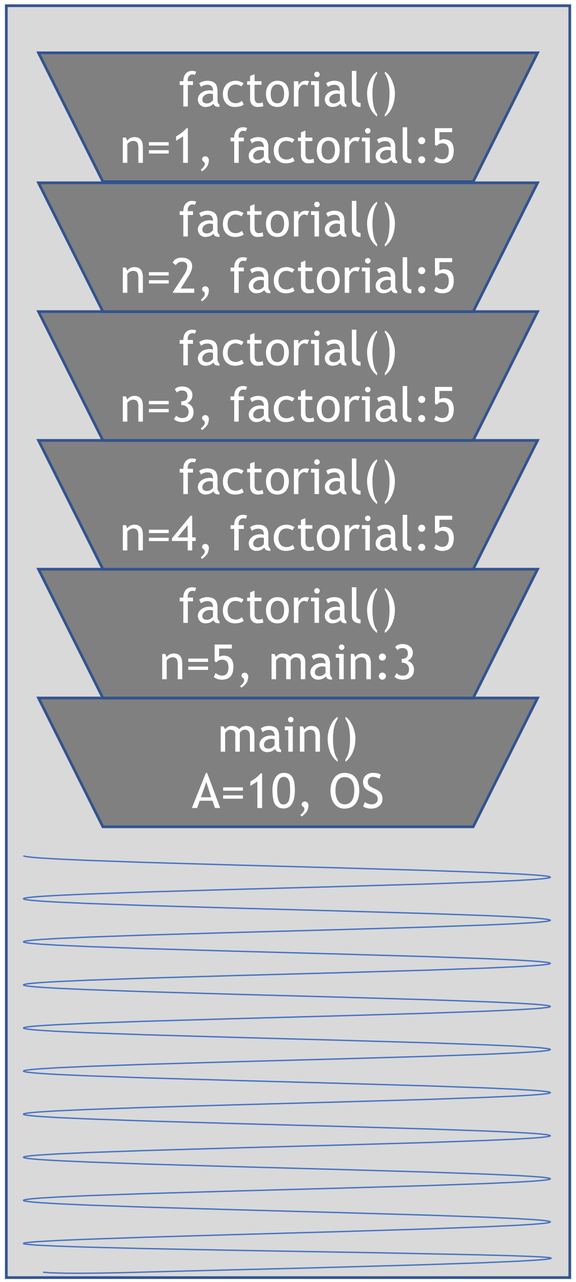
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=1) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

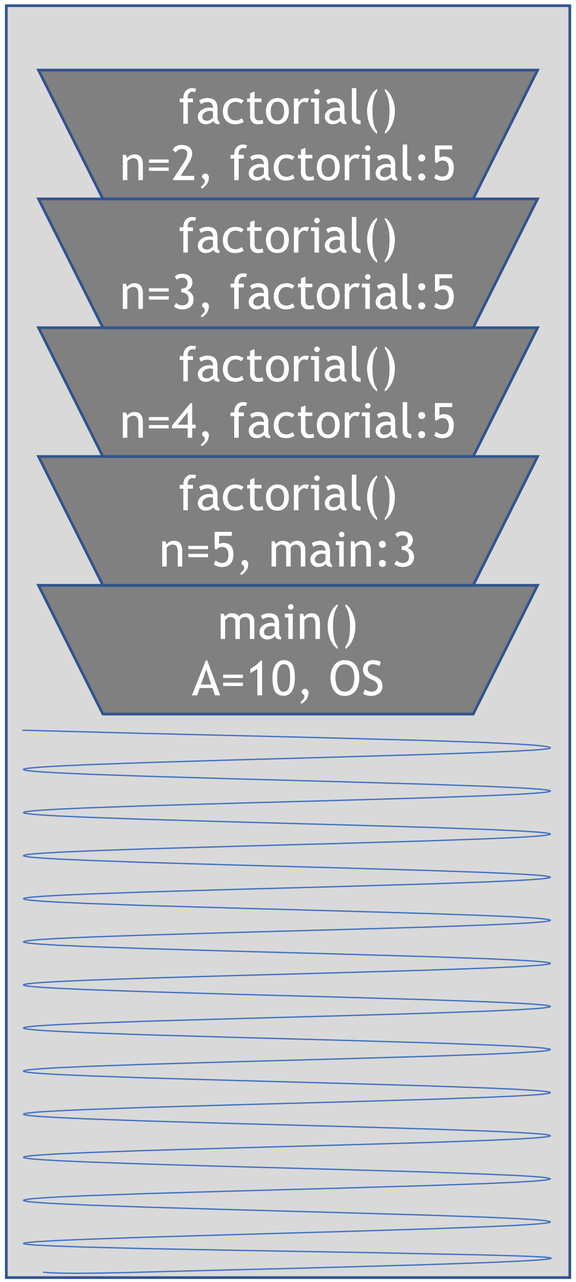
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=2) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n * 1;
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

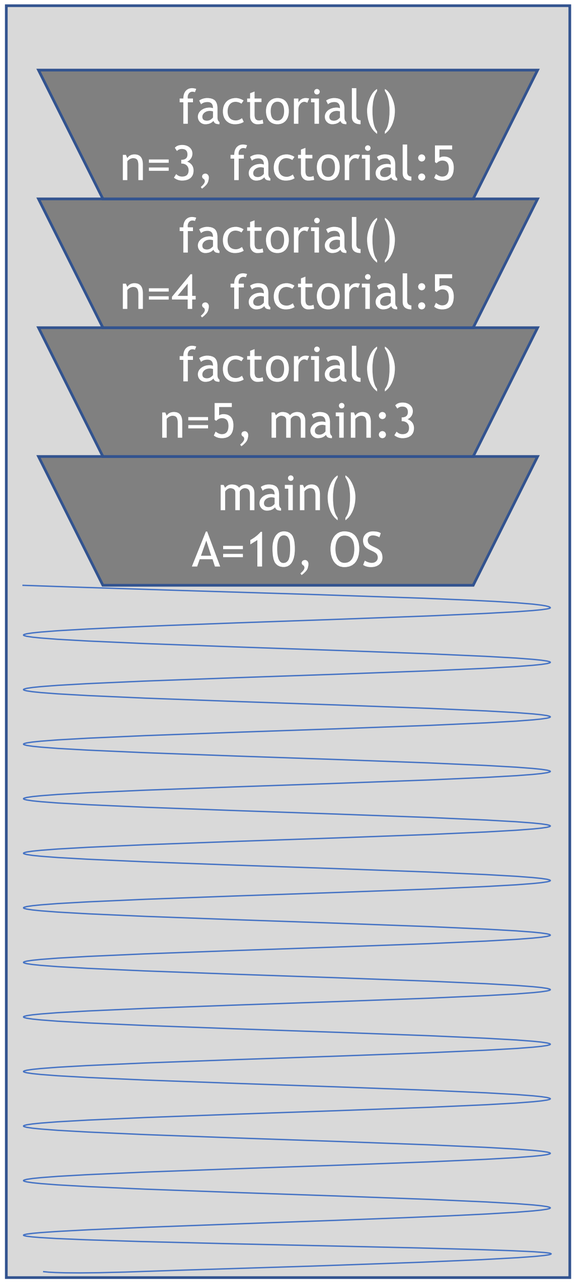
```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

```
1.   int factorial(int n) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n *
     factorial(n-1);
6.       return F;
7.     }
8.   }
```

Executing Function

```
1.   int factorial(int n=3) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n * 2;
6.       return F;
7.     }
8.   }
```

Call Stack



Compiled Code

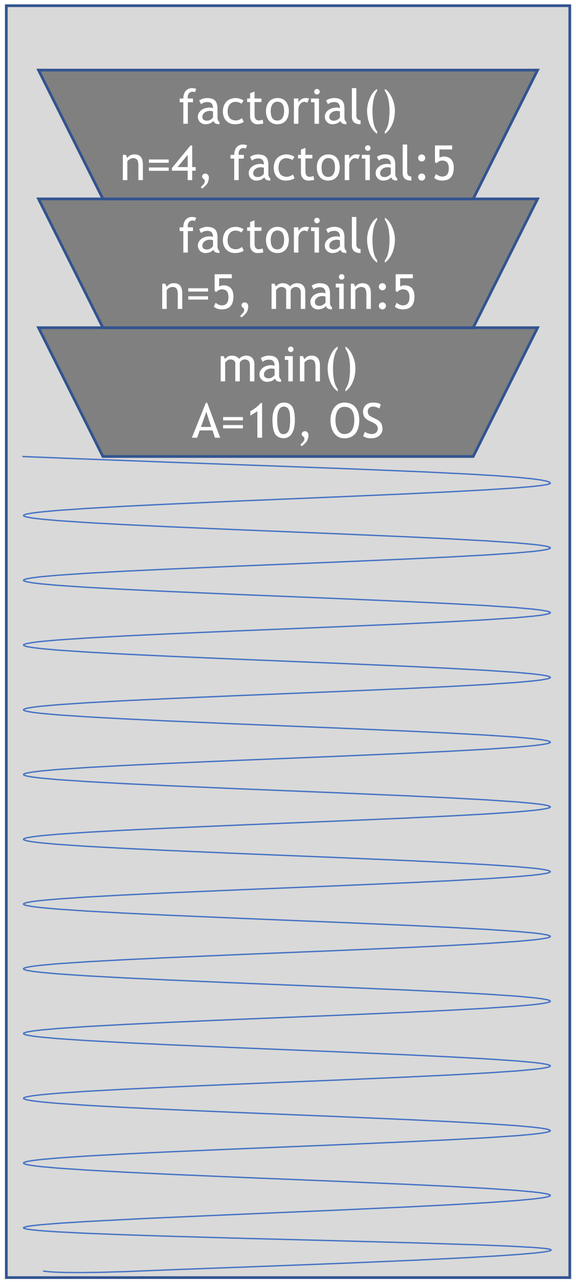
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=4) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n * 6;
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

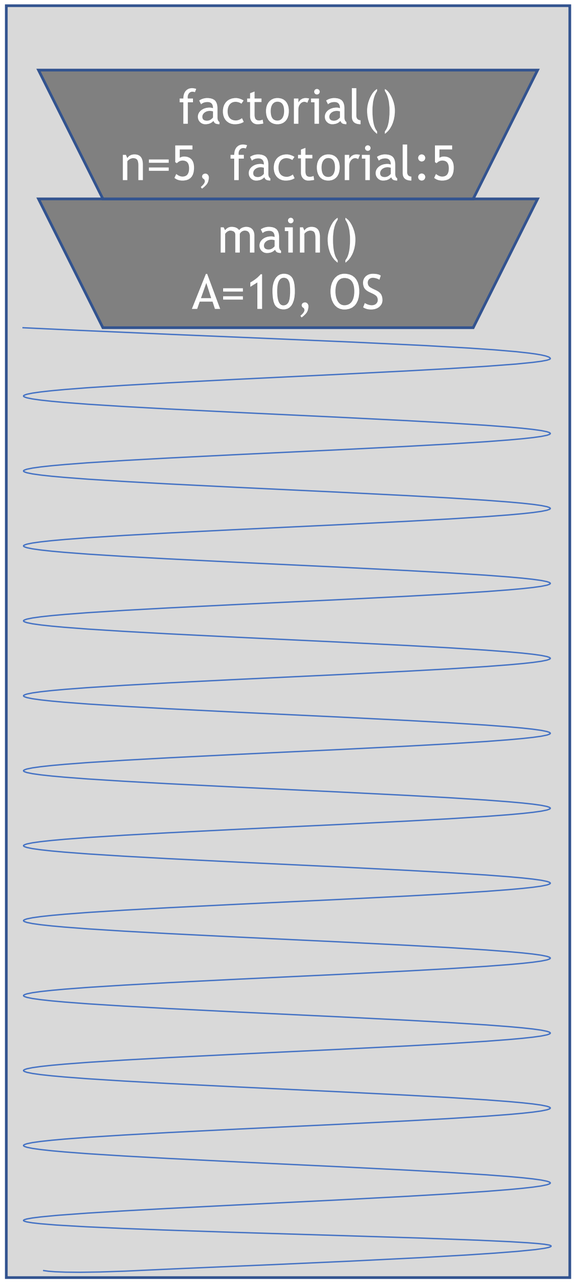
```
1.  void main() {
2.    int A = 10;
3.    int B = factorial(5);
4.    System.out.println(B);
5.  }
```

```
1.  int factorial(int n) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n *
    factorial(n-1);
6.      return F;
7.    }
8.  }
```

Executing Function

```
1.  int factorial(int n=5) {
2.    if (n == 1) {
3.      return 1;
4.    } else {
5.      int F = n * 24;
6.      return F;
7.    }
8.  }
```

Call Stack



Compiled Code

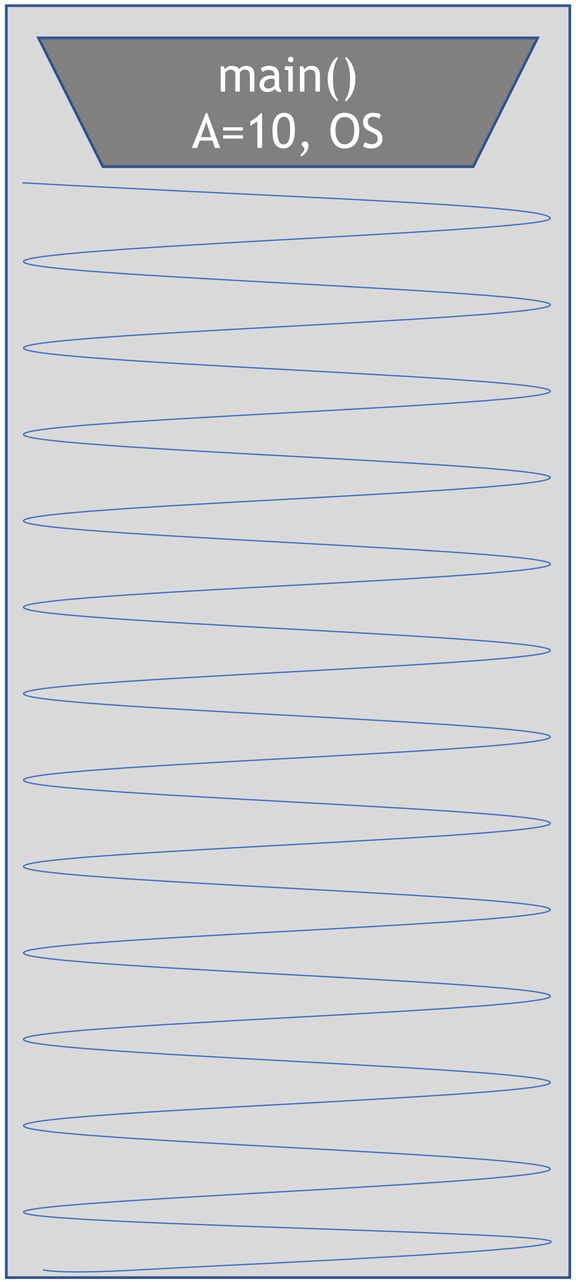
```
1.   void main() {
2.     int A = 10;
3.     int B = factorial(5);
4.     System.out.println(B);
5.   }
```

```
1.   int factorial(int n) {
2.     if (n == 1) {
3.       return 1;
4.     } else {
5.       int F = n *
     factorial(n-1);
6.       return F;
7.     }
8.   }
```

Executing Function

```
1.   void main() {
2.     int A = 10;
3.     int B = 120;
4.     System.out.println(B);
5.   }
```

Call Stack



Recursion

A method that calls itself, either directly or indirectly
Importantly, need a way to stop

```
public void badRecurse(int c)
{
  System.out.println("A" + c);
  badRecurse(c-1);
}
```

Class Recurser

```
public void goodRecurse(int c)
{
  System.out.println("B" + c);
  if (c<=0) return;
  goodRecurse(c-1);
}
```

Recursion — return values

```java
/**
 * A recursive function to add two positive numbers
 * @param num1 one of the numbers
 * @param num2 another number
 * @return the sum of the two numbers
 */
public int rAdder(int num1, int num2) {
    if (num2<=0)
        return num1;
    return rAdder(num1+1, num2-1);
}
public int rAdderB(int num1, int num2) {
    if (num2<=0)
        return 0;
    return 1+rAdderB(num1, num2-1);
}
```

Recursion — return values

```
    /**
     * Implement multiplication recursively using addition
     * For example, given the args 7 and 4 write a recusive function
     * that computes 7+7+7+7
     * @param i1 a number
     * @param i2 another number
     * @return i1*i2
     */
    public int multiply(int i1, int i2) {

    }
```

Recursion — returning values & private recursive functions

```
public BigInteger fibonacci(int n) {
        if (n<=0) return BigInteger.valueOf(0);
         if (n<3) return BigInteger.valueOf(1);
         return iFibonacci(BigInteger.valueOf(1), BigInteger.valueOf(1), n-2);
    }


private BigInteger iFibonacci(BigInteger fibNumA, BigInteger
fibNumB, int counter)
    {
         if (counter==1)
             return fibNumA.add(fibNumB);
         return iFibonacci(fibNumB, fibNumA.add(fibNumB),
counter-1);
    }
```

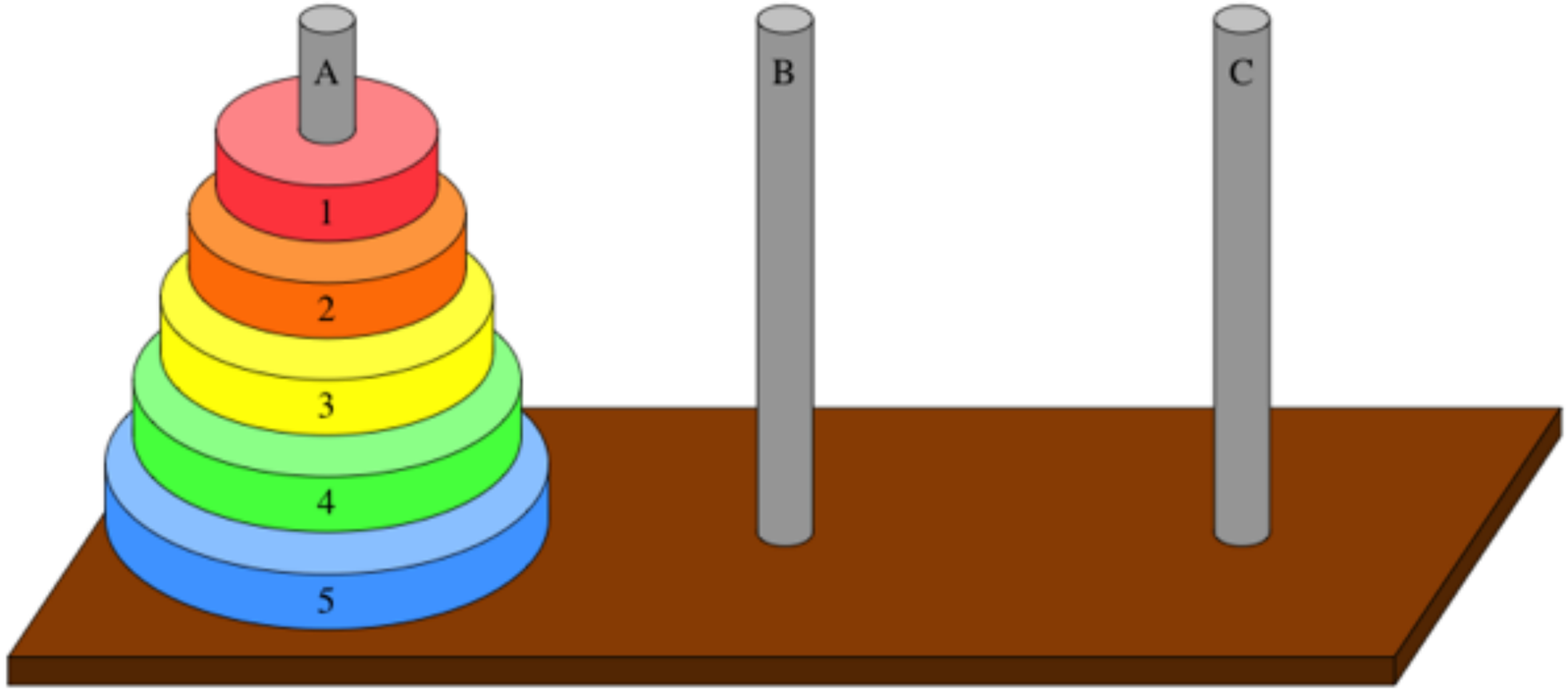
recursion practice

```
/**
 * Write a recursive function to add all the values in the array
 * Hint, this method should not be recursive.  Rather make a
 * private recursive function and call that from here
 * @param array
 * @return the sum of the numbers in the array
 */
public int addArray(int[] array);
```

more returning values

```
public ArrayList<Integer> rAccmulate(int count)
{
    if (count <= 0)
        return new ArrayList<Integer>();
    ArrayList<Integer> alAcc = rAccmulate(count-1);
    alAcc.add(count);
    return alAcc;
}
```

Towers of Hanoi



Complexity Analysis: $O(2^n)$!!!!