

CS151 Intro to Data Structures

Recursion

Binary Search

Announcements

- HW03 (Stacks & Queues) – due Friday 10/27
- Lab 04, 05, 06 due Friday 10/27
 - Lab 06 (today's lab) no checkoff, due on Gradescope

Outline

- Runtime
- Recursion
- Binary Search

Data Structure Operation Runtime

Construction/update/access operation times affect your run time

	Array	ArrayList	Linked list	ArrayStack	ArrayQueue
random access					
insert					
remove					
search					
min/max					

Data Structure Operation Time

Construction/update/access operation times affect your run time

	Array	ArrayList	Linked list	ArrayStack	ArrayQueue
random access	$O(1)$	$O(1)$	$O(n)$	-	-
insert	$O(1)^*$ or $O(n)$	$O(1)^*$ or $O(n)$	$O(1)$	$O(1)^*$	$O(1)^*$
remove	$O(1)^*$ or $O(n)$	$O(1)^*$ or $O(n)$	$O(1)$	$O(1)^*$	$O(1)^*$
search	$O(n)$	$O(n)$	$O(n)$	-	-
min/max	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Dynamic Array

Array is replaced with a larger one when `add` is performed on full

- Allocate a new larger array
- Copy all existing elements into the beginning of new array

How much bigger?

- incremental: increase size by a constant c
- doubling: double the size

Amortized Analysis

Because copying is needed, every time we replace the array it costs $O(n)$

However, we do not need to expand every time we call `add`

Amortized time is:

the average time taken over a series of n operations

$T(n)/n$ - account for uneven distribution of work

Start with an array of size 1

Incremental Analysis

Over n `add` operations, we replace the array $k = \frac{n}{c}$ times, each time adding c

Let $T(n)$ denote the time of n `add` ops

$$\begin{aligned} T(n) &= n + c + 2c + \dots + (k - 1)c \\ &= n + (1 + 2 + \dots + k - 1)c = n + \frac{(k - 1)k}{2}c \\ &= O(n + k^2) = O(n^2) \end{aligned}$$

`add` with incremental is $\frac{O(n^2)}{n} = O(n)$

Doubling Analysis

Over n add operations, we replace the array $k = \log n$ times

Let $T(n)$ denote the time of n add ops

$$\begin{aligned} T(n) &= n + 1 + 2 + 4 \dots + 2^{k-1} \\ &= n + 2^k - 1 = n + 2^{\log n} - 1 \\ &= n + 2^{\log n} - 1 = 2n - 1 \\ &= O(n) \end{aligned}$$

add with doubling is $\frac{O(n)}{n} = O(1)$

Data Structure Operation Time

Construction/update/access operation times affect your run time

	Array	ArrayList	Linked list	ArrayStack	ArrayQueue
random access	$O(1)$	$O(1)$	$O(n)$	-	-
insert	$O(1)^*$ or $O(n)$	$O(1)^*$ or $O(n)$	$O(1)$	$O(1)^*$	$O(1)^*$
remove	$O(1)^*$ or $O(n)$	$O(1)^*$ or $O(n)$	$O(1)$	$O(1)^*$	$O(1)^*$
search	$O(n)$	$O(n)$	$O(n)$	-	-
min/max	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Outline

- Runtime
- **Recursion**
- Binary Search

The Factorial

- Recursive definition:

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

- Java method

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

The Factorial

- Recursive definition:

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

- Java method

```
1  public static int factorial(int n) throws IllegalArgumentException {  
2      if (n < 0)  
3          throw new IllegalArgumentException();    // argument must be nonnegative  
4      else if (n == 0)  
5          return 1;                               // base case  
6      else  
7          return n * factorial(n-1);              // recursive case  
8  }
```

Recursive Method

Break problem down into smaller subproblem that we can repeat

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
- Often called “the rule”

Compiled Code

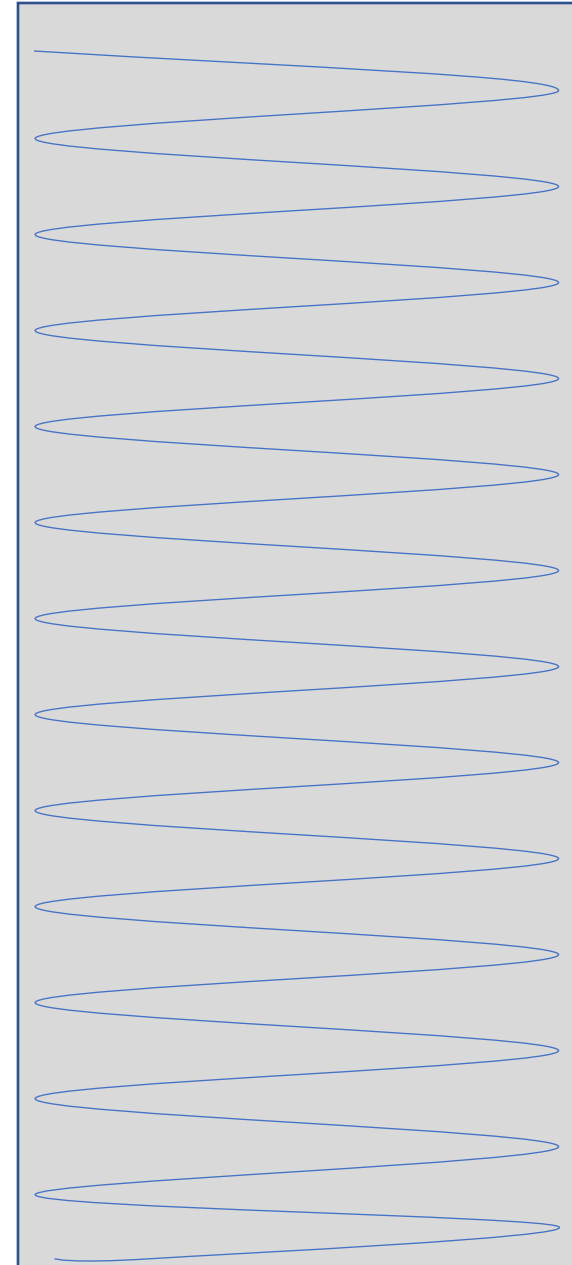
```
1. void main() {  
2.     int A = 10;  
3.     int B = factorial(5);  
4.     System.out.println(B);  
5. }
```

```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Executing Function



Call Stack




Compiled Code

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

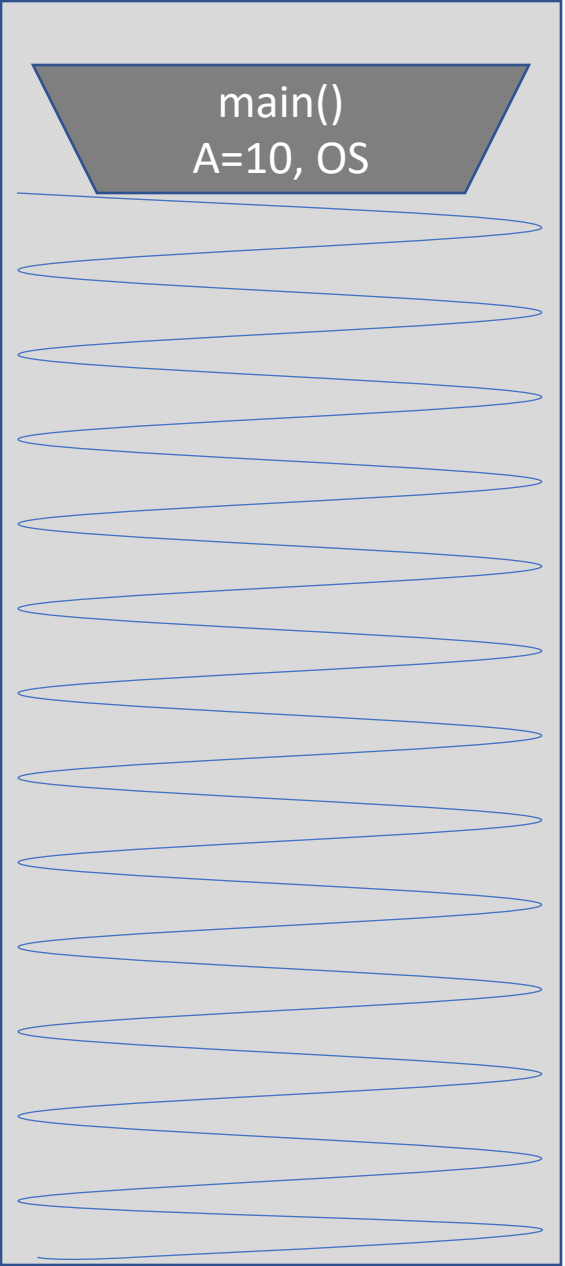
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function



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

Call Stack



main()
A=10, OS


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 *  
6.             factorial(n-1);  
7.     }  
8. }
```

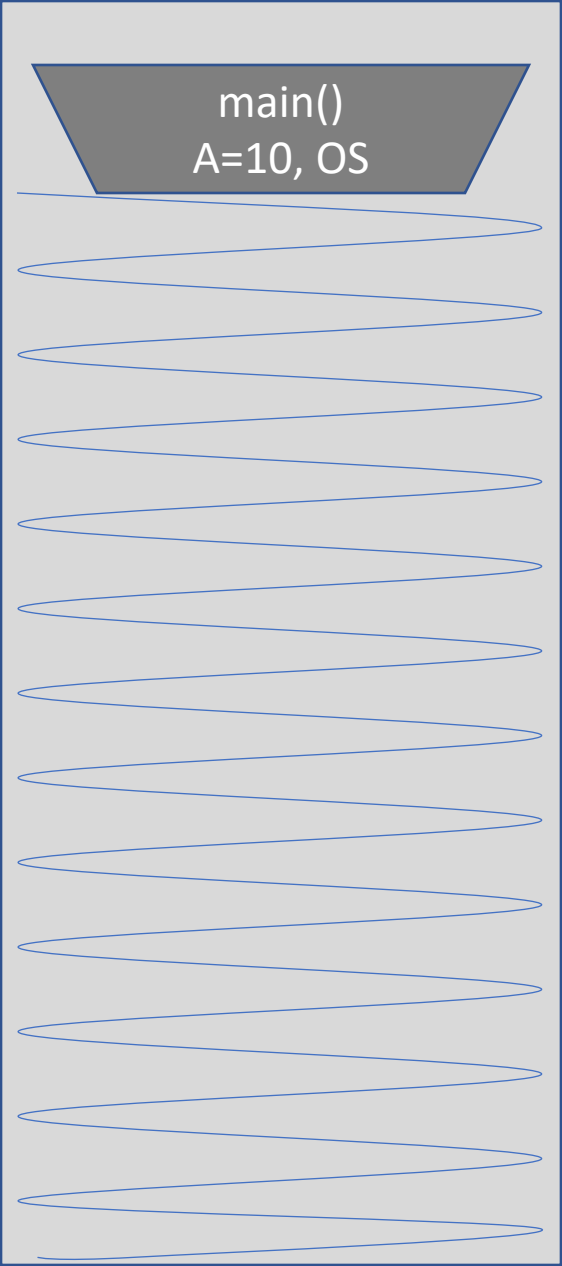
Executing Function

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



Call Stack

main()
A=10, OS




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 *  
6.             factorial(n-1);  
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 *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Executing Function



```
1. int factorial(int n=5) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Call Stack




Compiled Code

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

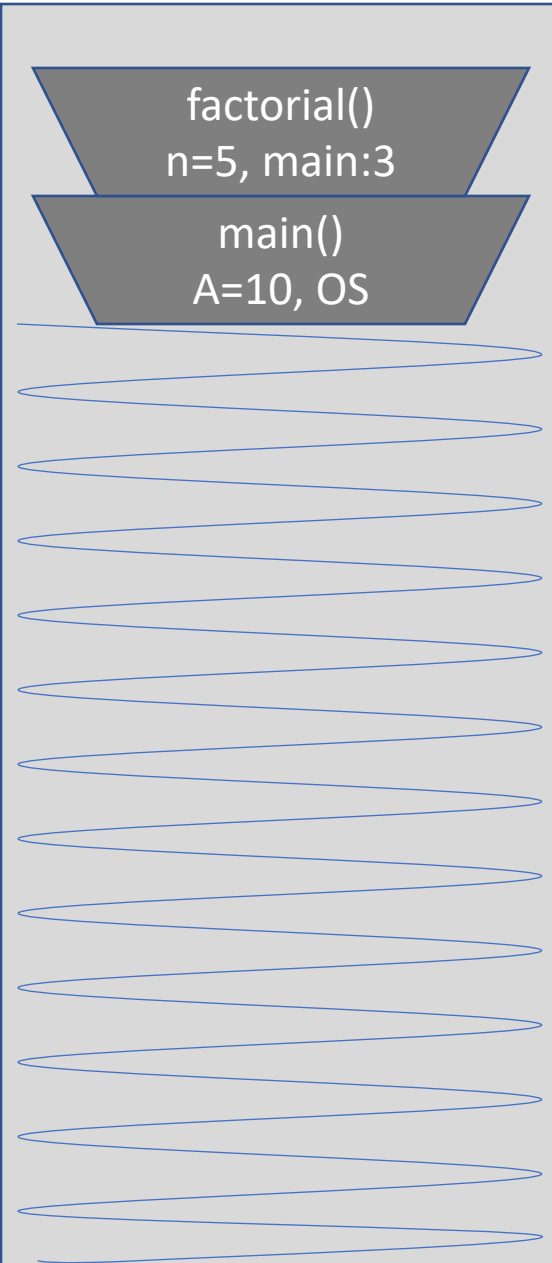
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack




Compiled Code

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

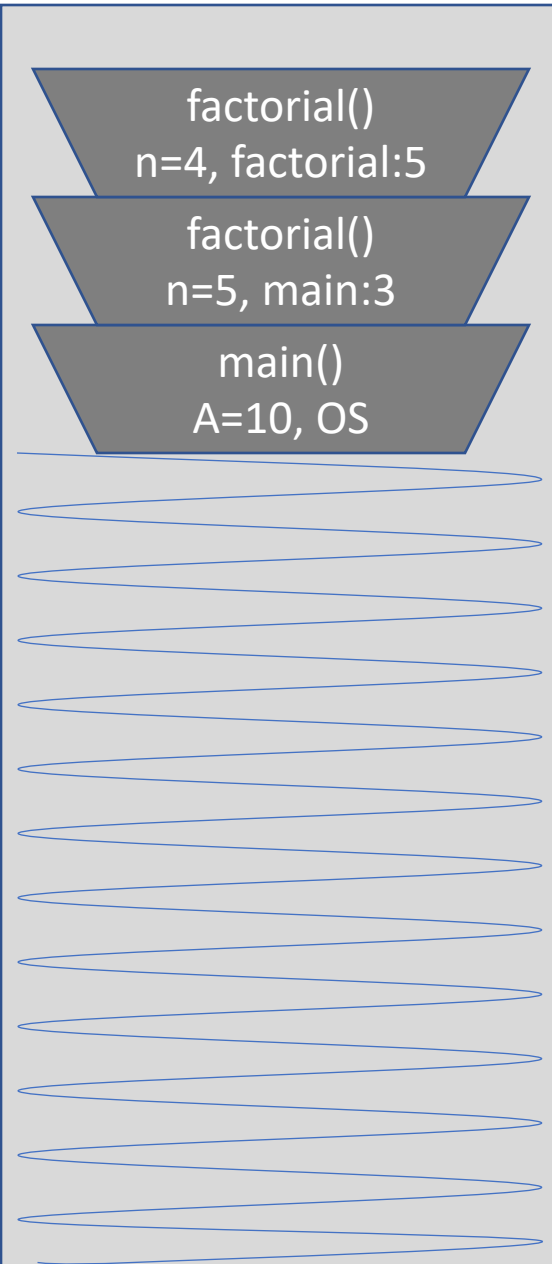
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack




Compiled Code

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

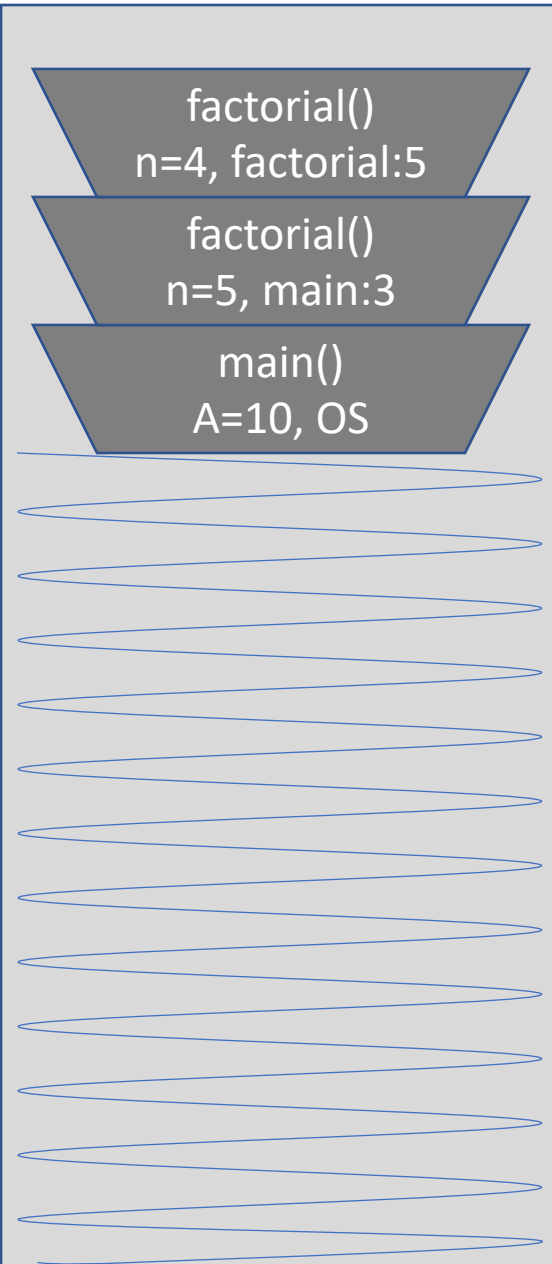
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Executing Function



```
1. int factorial(int n=4) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Call Stack




Compiled Code

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

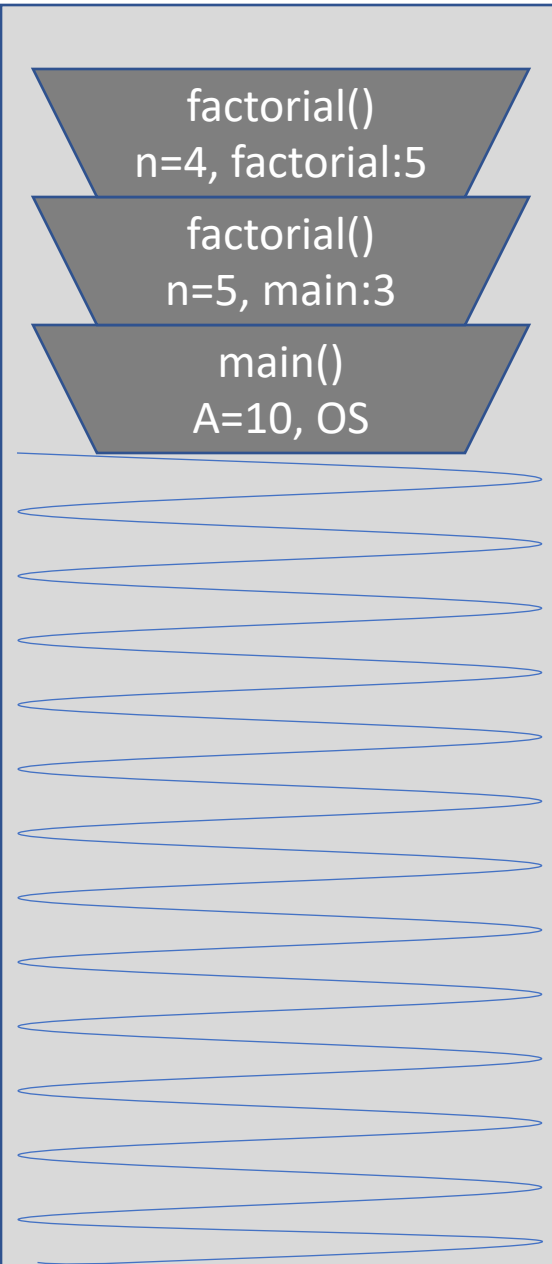
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack




Compiled Code

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

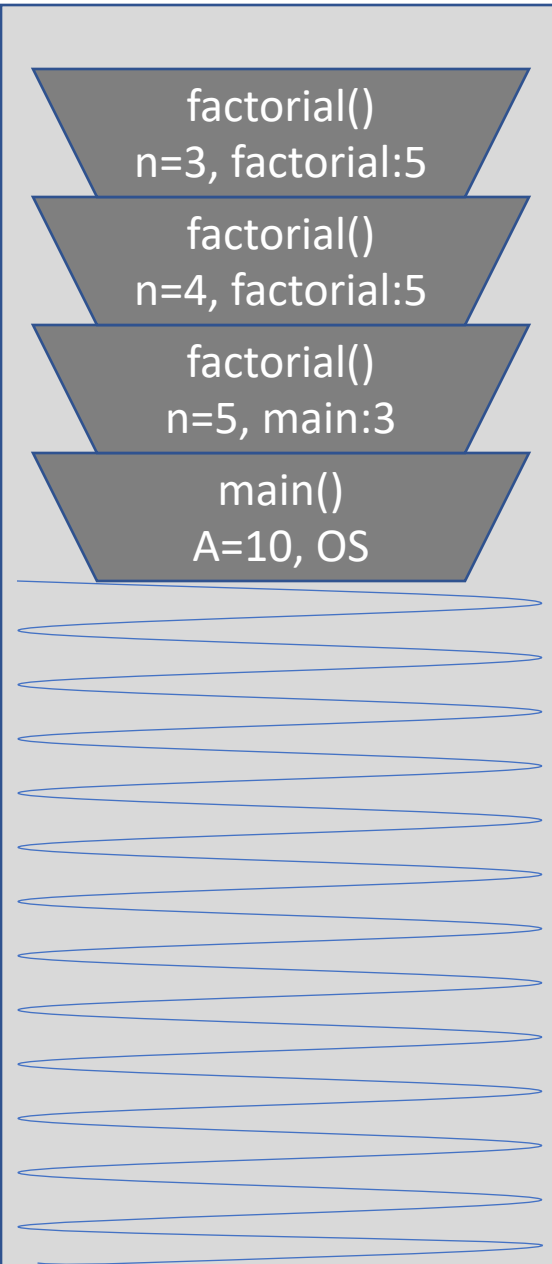
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack




Compiled Code

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

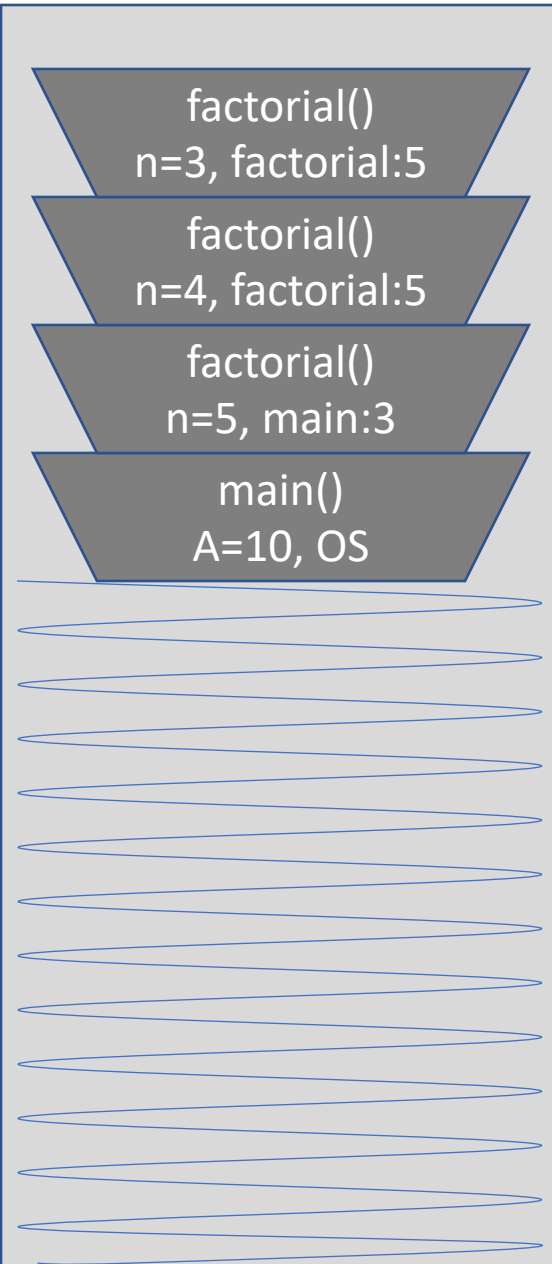
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Executing Function



```
1. int factorial(int n=3) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Call Stack




Compiled Code

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

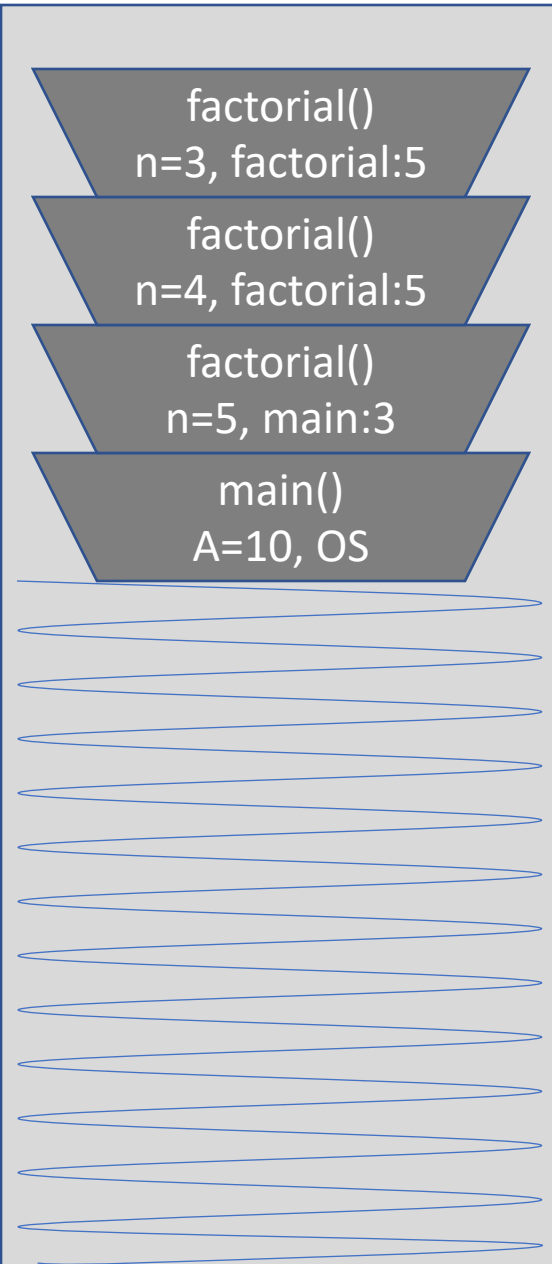
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack




Compiled Code

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

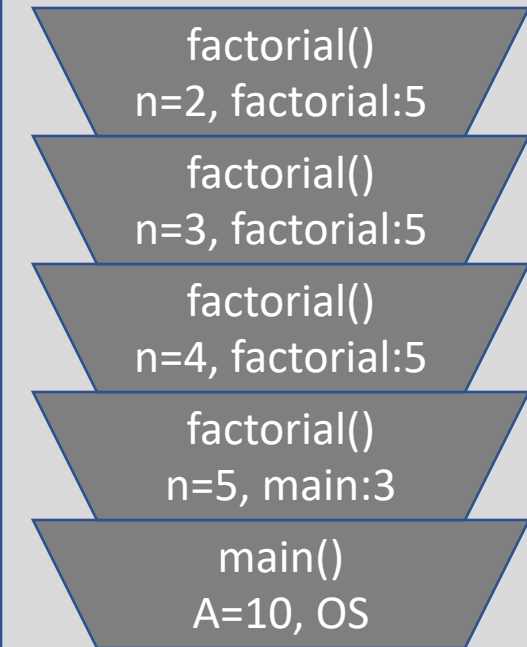
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack




Compiled Code

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

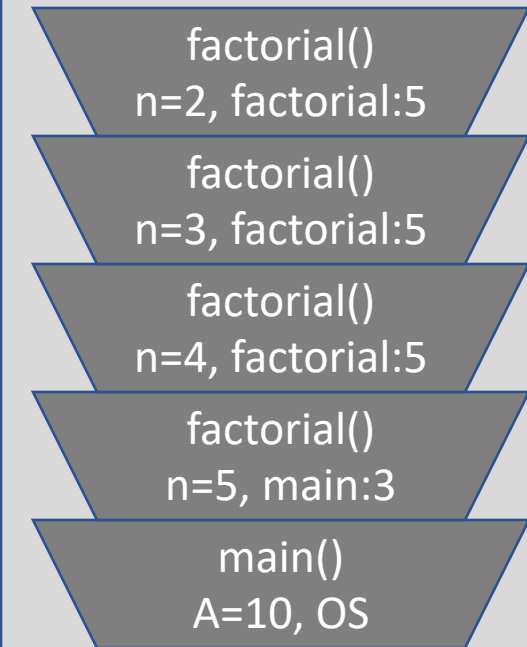
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Executing Function



```
1. int factorial(int n=2) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Call Stack




Compiled Code

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

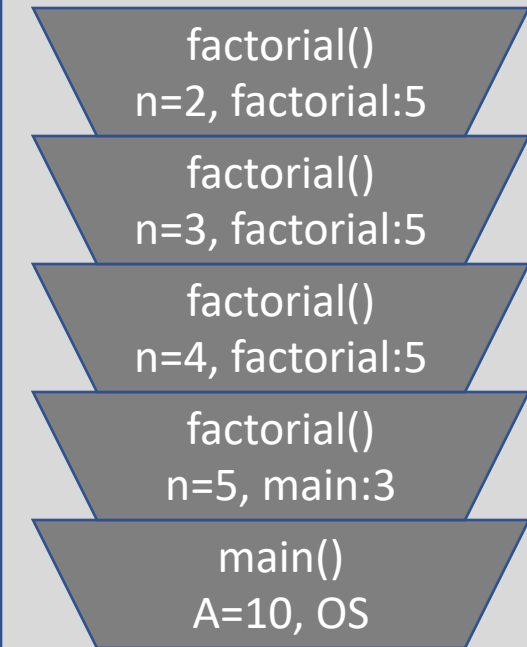
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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



Call Stack



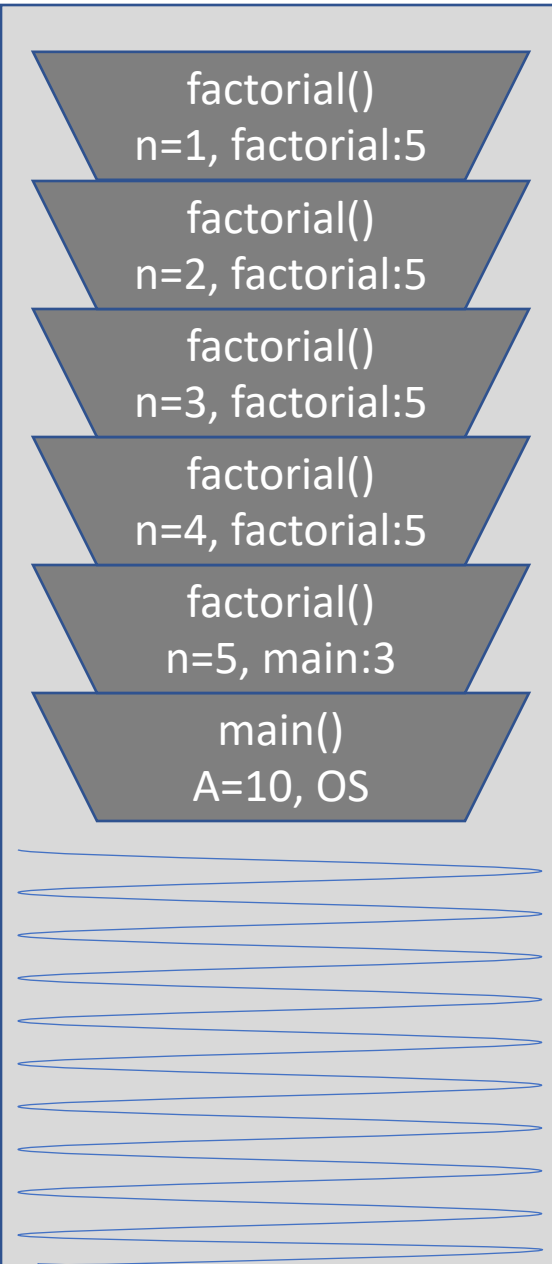

Compiled Code

```
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 *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function

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




Compiled Code

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

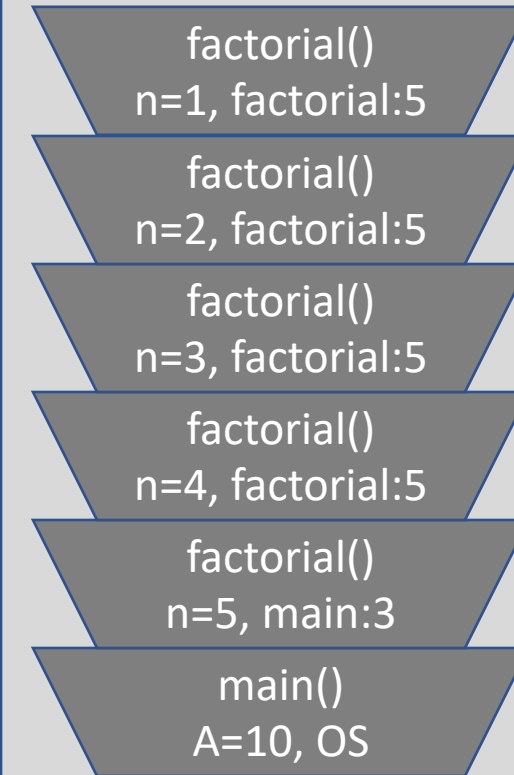
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.     }  
8. }
```

Executing Function



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

Executing Function

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

Call Stack

factorial()
n=1, factorial:5

factorial()
n=2, factorial:5

factorial()
n=3, factorial:5

factorial()
n=4, factorial:5

factorial()
n=5, main:3

main()
A=10, OS


Compiled Code

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

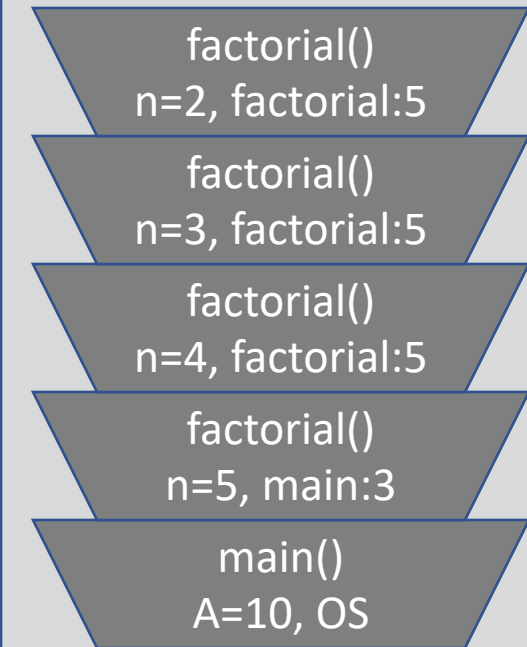
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

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. }
```

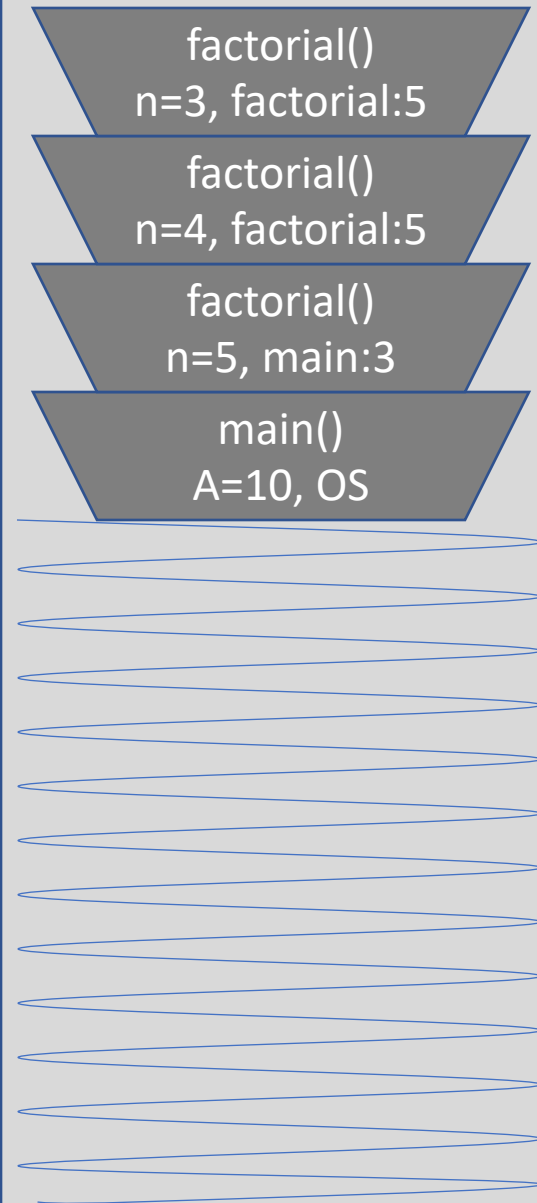
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

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. }
```

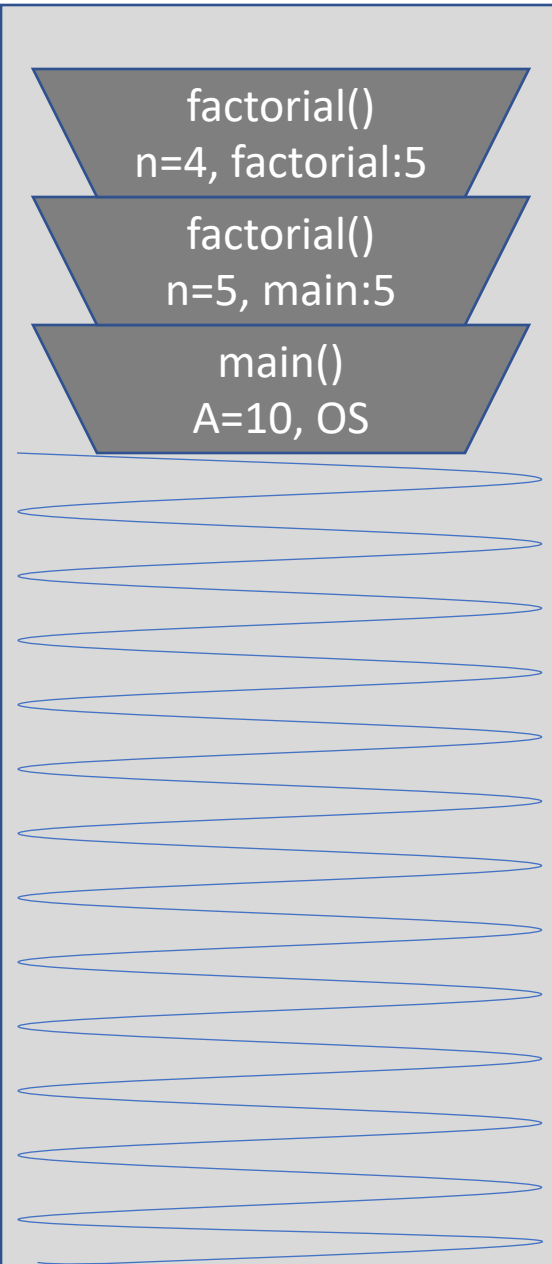
```
1. int factorial(int n) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

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 *  
6.             factorial(n-1);  
7.         return F;  
8.     }  
9. }
```

Executing Function

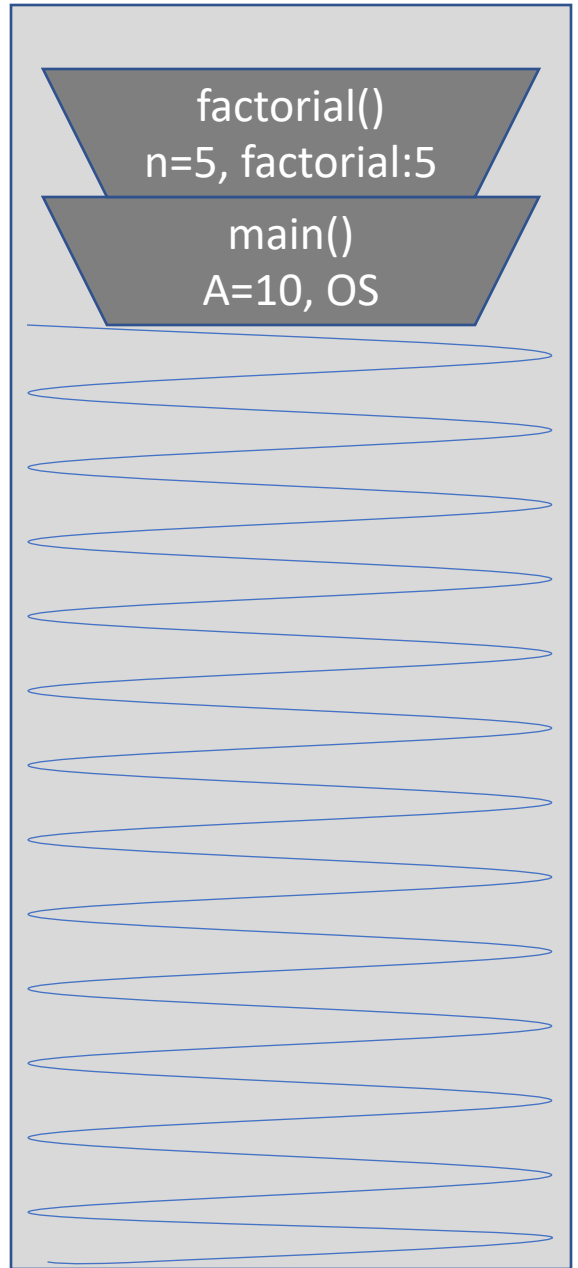
```
1. int factorial(int n=5) {  
2.     if (n == 1) {  
3.         return 1;  
4.     } else {  
5.         int F = n * 24;  
6.         return F;  
7.     }  
8. }
```



Call Stack

factorial()
n=5, factorial:5

main()
A=10, OS




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 *  
6.             factorial(n-1);  
7.     }  
8. }
```

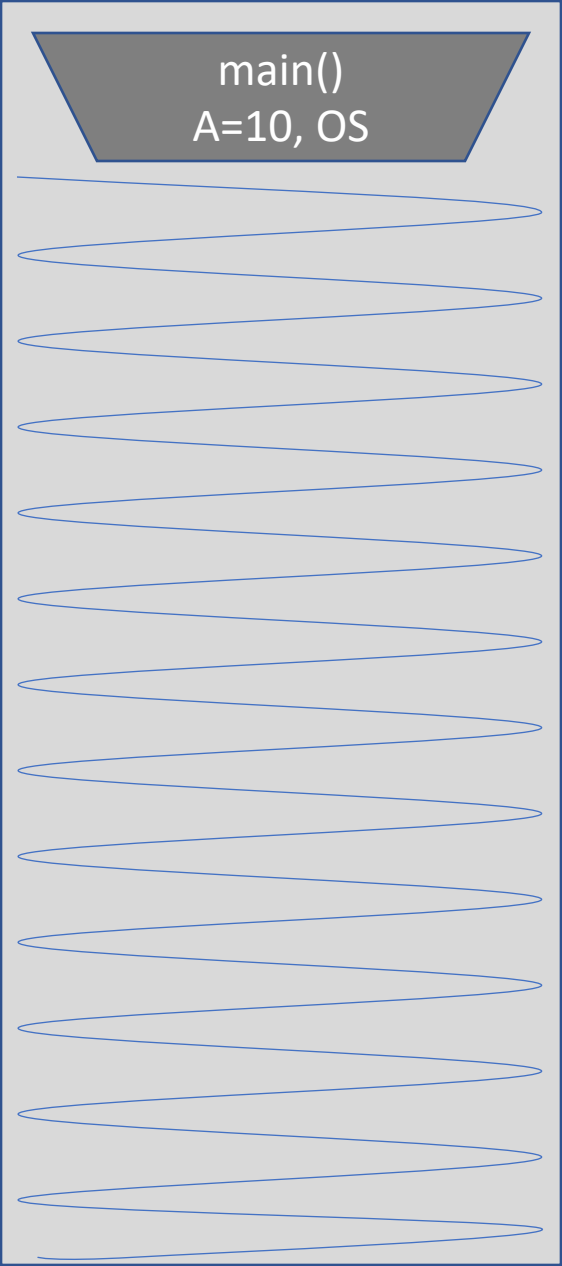
Executing Function

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



Call Stack

main()
A=10, OS



Outline

- Runtime
- Recursion
- **Binary Search**

Binary Search

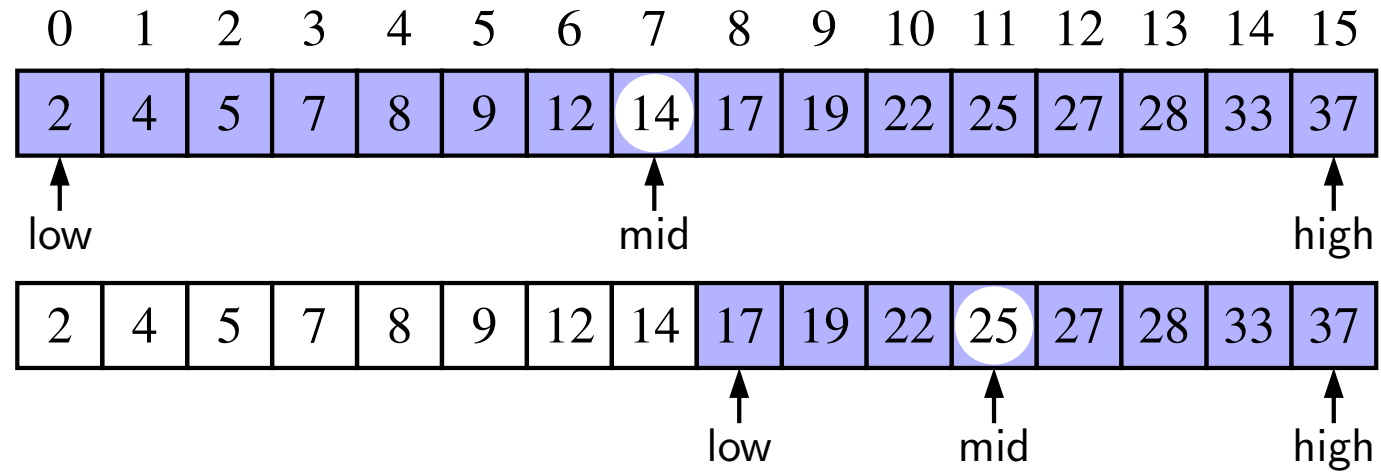
Search for an integer (22) in an ordered list

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

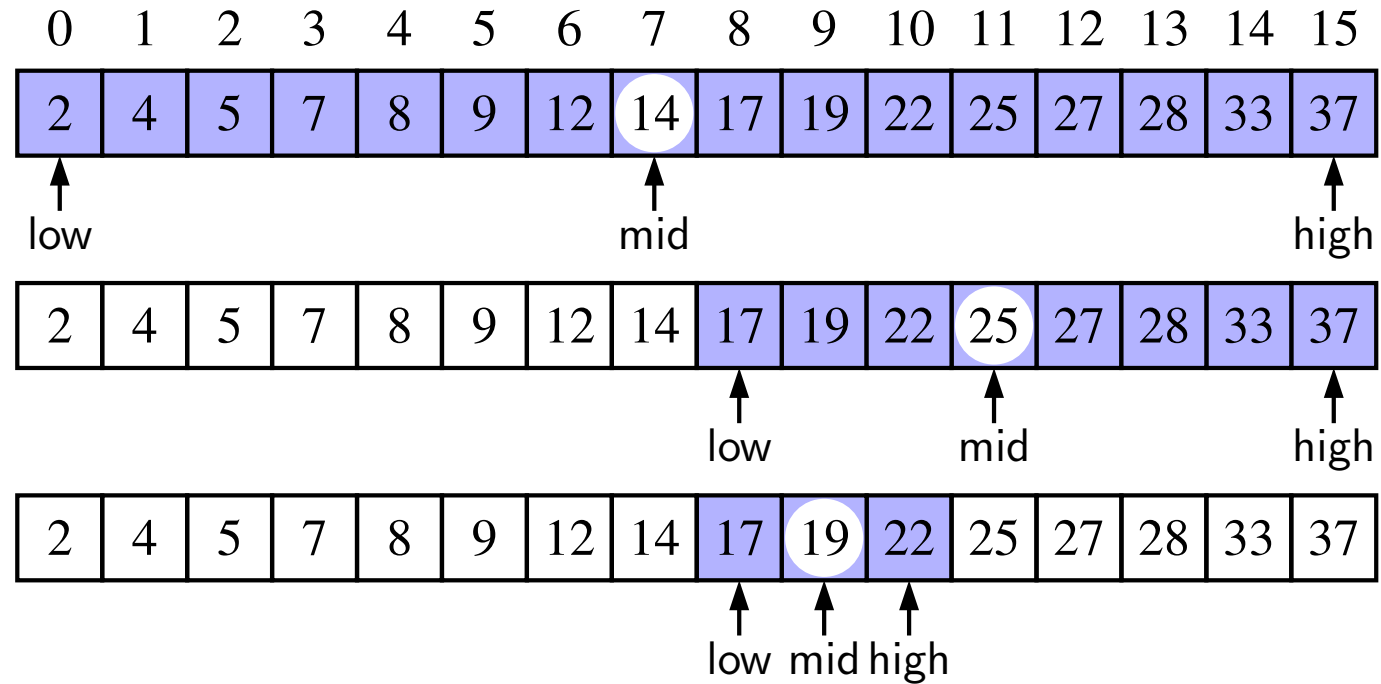
target = 22

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37
↑ low							↑ mid								↑ high

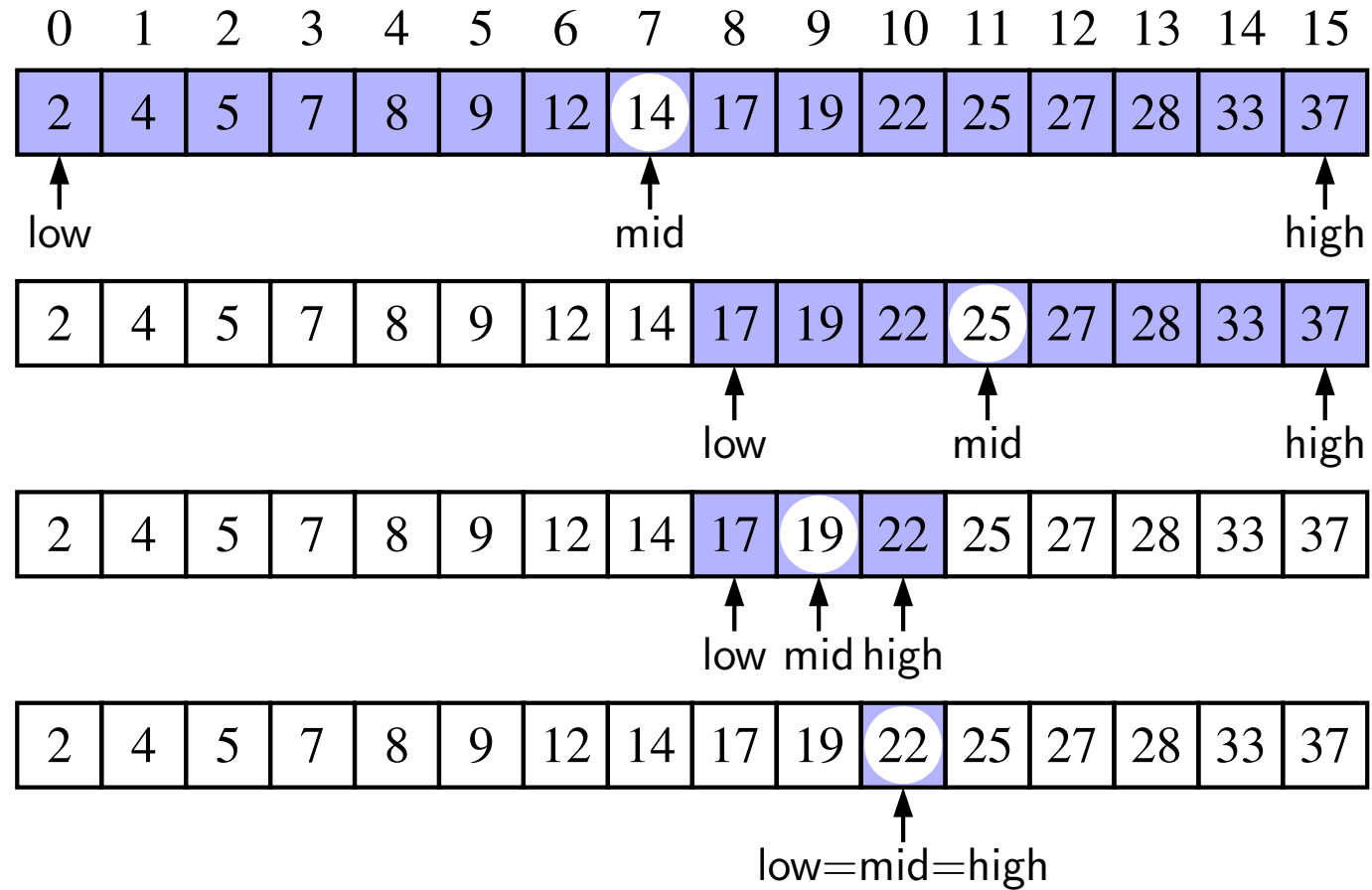
target = 22



target = 22



target = 22



Binary Search

Search for an integer (22) in an ordered list

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor = \left\lfloor \frac{0 + 15}{2} \right\rfloor = 7$$

- `target == data[mid]`, found
- `target > data[mid]`, recur on second half
- `target < data[mid]`, recur on first half

Code

```
1  /**
2   * Returns true if the target value is found in the indicated portion of the data array.
3   * This search only considers the array portion from data[low] to data[high] inclusive.
4   */
5  public static boolean binarySearch(int[ ] data, int target, int low, int high) {
6      if (low > high)
7          return false;                // interval empty; no match
8      else {
9          int mid = (low + high) / 2;
10         if (target == data[mid])
11             return true;              // found a match
12         else if (target < data[mid])
13             return binarySearch(data, target, low, mid - 1); // recur left of the middle
14         else
15             return binarySearch(data, target, mid + 1, high); // recur right of the middle
16     }
17 }
```

Binary Search Analysis

Each recursive call divides the array in half

If the array is of size n , it divides (and searches) at most $\log n$ times before the current half is of size 1

$O(\log n)$

Comparable

Binary search on a list of objects requires that the objects have natural ordering

In other words, the objects must implement Comparable

DS Operation Times

Data structure choices affect your run time

	Unsorted array	Sorted array	Unsorted list	Sorted list
random access	$O(1)$	$O(1)$	$O(n)$	$O(n)$
insert	$O(1)^*$ or $O(n)$	$O(n)$	$O(1)$	$O(n)$
remove	$O(1)^*$ or $O(n)$	$O(1)^*$ or $O(n)$	$O(1)$	$O(1)$
search	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
min/max	$O(n)$	$O(1)$	$O(n)$	$O(1)$

DS Operation Times

Data structure choices affect your run time

	Unsorted array	Sorted array	Unsorted list	Sorted list
random access	$O(1)$	$O(1)$	$O(n)$	$O(n)$
insert	$O(1)^*$ or $O(n)$	$O(n)$	$O(1)$	$O(n)$
remove	$O(1)^*$ or $O(n)$	$O(1)^*$ or $O(n)$	$O(1)$	$O(1)$
search	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
min/max	$O(n)$	$O(1)$	$O(n)$	$O(1)$