CS151 Intro to Data Structures

Iterators
Recursion

Binary Search

Announcements

- HW03 due tomorrow
- Lab 4 and 5 due dates?
  - Lab 4 was interfaces. Due tomorrow...
  - Lab 5 was stacks. Related to your homework but due next thursday?
  - Next two labs will not be checked off. They're just a head start on your homework.

Outline

- Iterators
- Runtime
- Recursion
- Binary Search

ArrayList ADT

Whats an ADT?

ArrayList

Big-O memory?

- O(n)

Indexing / random access?

- O(1)

Add / remove?

- O(n)

Iterators

Iterators

- represents a sequence of elements and provides a way to iterate, or traverse, through those elements one at a time

Iterators

- Abstracts the process of scanning through a sequence of elements (traversal)
- provides a way to iterate, or traverse, through elements one at a time

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.

next( ): Returns the next element in the sequence.

- Combination of these two methods allow a generic traversal structure

```
while(iter.hasNext()) {
  iter.next();
}
```

Iterators

- **code**

- Can an iterator go backwards? NO. Only can do `next()`

`Iterable` Interface

- What can i use an `iterator` on? Anything that implements the `iterable` interface.
- Each call to `iterator()` returns a new iterator instance, thereby allowing traversals of a collection
- `List` interface extends `Iterable` and `ArrayList` implements `List`

`Iterable` Interface

An interface with a single method:

- `iterator()`: returns an iterator of the elements in the collection

`Iterator` Interface

Iterat**or** Interface

Another interface that supports iteration

- boolean hasNext()
- E next()
- void remove()

- Scanner implements Iterator<String>
- ArrayList inner class ArrayListIterator implements Iterator

Let’s make ExpandableArray iterable

`Iterable` versus `Iterator`?

- `Iterable`
  - `java.lang`
  - override `iterator()`
  - Doesn't store the iteration state
  - Removing elements during iteration isn't allowed

- `Iterator`
  - `java.util`
  - Override `hasNext(),next()`
  - Optional `remove()`
  - Stores iteration state (list cursor)
  - Removing elements during iteration supported

Iterators Review

Iterators

- represents a sequence of elements and provides a way to iterate, or traverse, through those elements one at a time

Iterators

- Abstracts the process of scanning through a sequence of elements (traversal)
- provides a way to iterate, or traverse, through elements one at a time

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.

next( ): Returns the next element in the sequence.

- Combination of these two methods allow a generic traversal structure

```
while(iter.hasNext()) {
  iter.next();
}
```

Iterators

Can an iterator go backwards? NO. Only can do `next()`

`Iterable` Interface

An interface with a single method:

- `iterator()`: returns an iterator of the elements in the collection

`Itera`**`t`**<u>**`or`**</u> Interface

Another interface that supports iteration

- `boolean hasNext()`
- `E next()`
- `void remove()`

- `Scanner` implements `Iterator<String>`
- `ArrayList` inner class `ArrayListIterator` implements `Iterator`

Iterable Expandable Array

`Iterable` versus `Iterator`?

- `Iterable`
  - `java.lang`
  - override `iterator()`
  - Doesn't store the iteration state
  - Removing elements during iteration isn't allowed
- `Iterator`
  - `java.util`
  - Override `hasNext(),next()`
  - Optional `remove()`
  - Stores iteration state (list cursor)
  - Removing elements during iteration supported

Runtime Analysis Review

Data Structure Operation Runtime

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

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

The worst case is unlikely to occur

Amortized: the average run time over a series of operations

Accounts for an uneven distribution of work

Amortized Analysis of an Expandable Array

When the array is full, we can have two expansion strategies

- expand the array by doubling the size
  - `new_arr[numElems*2]`
  - “doubling expansion”

- expand the array by a constant `c`
  - `new_arr[numElems+c]`
  - “incremental expansion”

Amortized Analysis of "Doubling Expansion"

Example: start with an array of size `1`

Let's compute two things:

1. the number of times we need to expand: `k(n)`
2. the total number operations: `T(n)`

`k(8) = 3`

`k(n) = logn`

`T(8) = 1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 = 15`

`T(n) = n` (as n approaches infinity)

Amortized `T(n)`?

`O(n)/n = O(1)`

Amortized Analysis of "Incremental Expansion"

Example: start with an array of size `1` and expand with `c=2`

Let's compute two things:

1. the number of times we need to expand: `k(n,c)`
2. the total number operations: `T(n)`

`k(6,2)= 3`

`k(n,c) = n/c`

`T(6) = 1 + 2 + 1 + 4 + 1 + 6 = 15`

`T(n) = O(n^2)` (as n approaches infinity)

Amortized T(n)?

`O(n^2)/ n = O(n)`

Outline

- Runtime
- **Recursion**
- Binary Search

Recursive functions – base case

Conditional statement that prevents infinite repetitions

Usually handles cases where:

- input is empty
- problem is at its smallest size

Recursion Example - Factorial

- What is a factorial? n!
- product of all integers less than or equal to n
  - n! = n * n-1 * n-2 1
  - 5! = 5 * 4 * 3 * 2 * 1
  - 4! = 4 * 3 * 2 * 1
  - 3! = 3 * 2 * 1

Visualizing recursion – Factorial example

factorial(5) =

= 5 * factorial(4)

= 5 * 4 * factorial(3)

= 5 * 4 * 3 * factorial(2)

= 5 * 4 * 3 * 2 * factorial(1)

= 5 * 4 * 3 * 2 * 1

Recursion Example – Contains letter

Write a method called “containsLetter” that determines if a String contains a given character

Question: What are the parameters?

1. The character to look for
2. The string to be looking in

Question: What is the return type?

Code it!

Recursion Visualization – Contains letter

contains(“l”, “apple”) =

contains(“l”, “apple”)

contains(“l”, “pple”)

contains(“l”, “ple”)

contains(“l”, “le”)

return true

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

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

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

Executing Function

Call Stack

Compiled Code

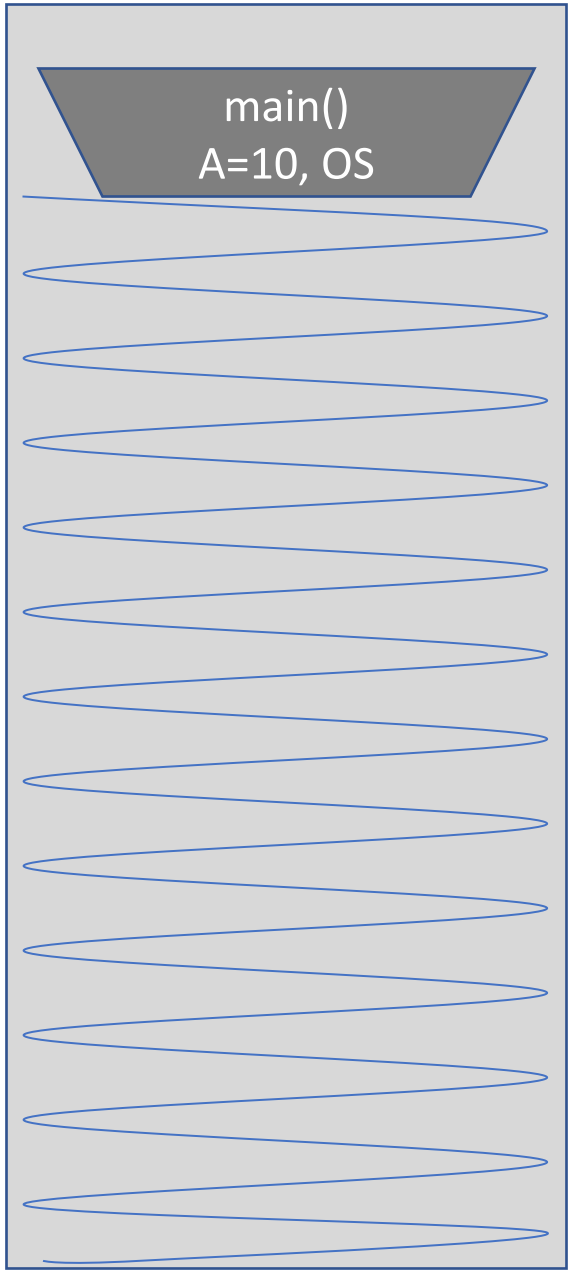
```
void main() {
  int A = 10;
  int B = factorial(5);
  System.out.println(B);
}
```

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

Executing Function

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

Call Stack



Compiled Code

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

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

Executing Function

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

Call Stack

main()
A=10, OS

Compiled Code

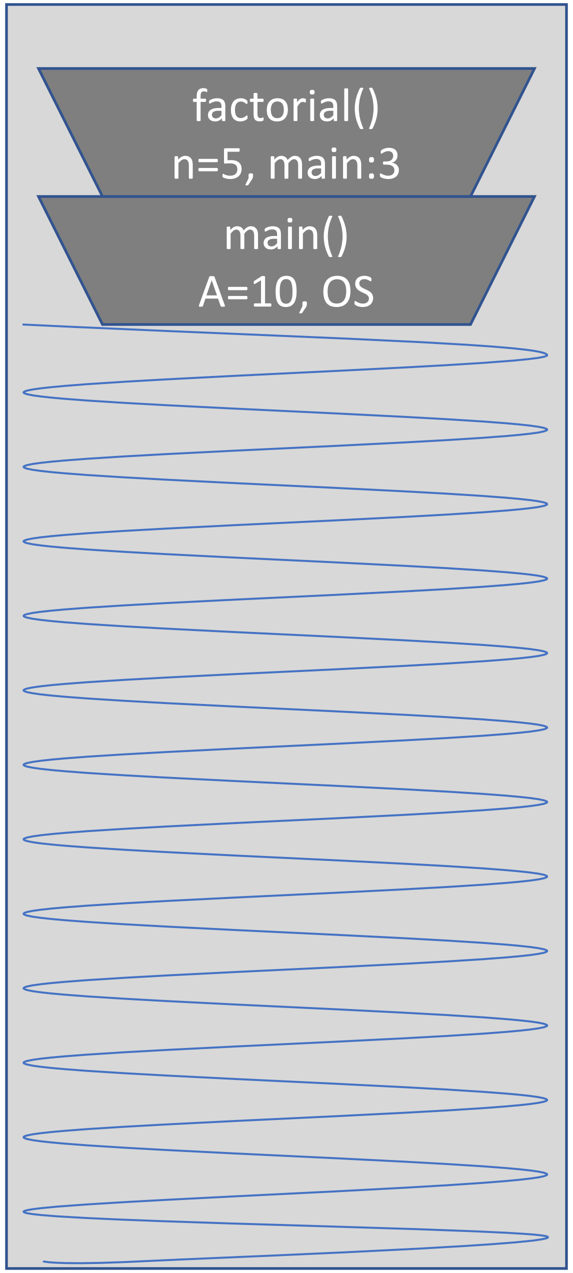
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

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

Executing Function

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

Call Stack



Compiled Code

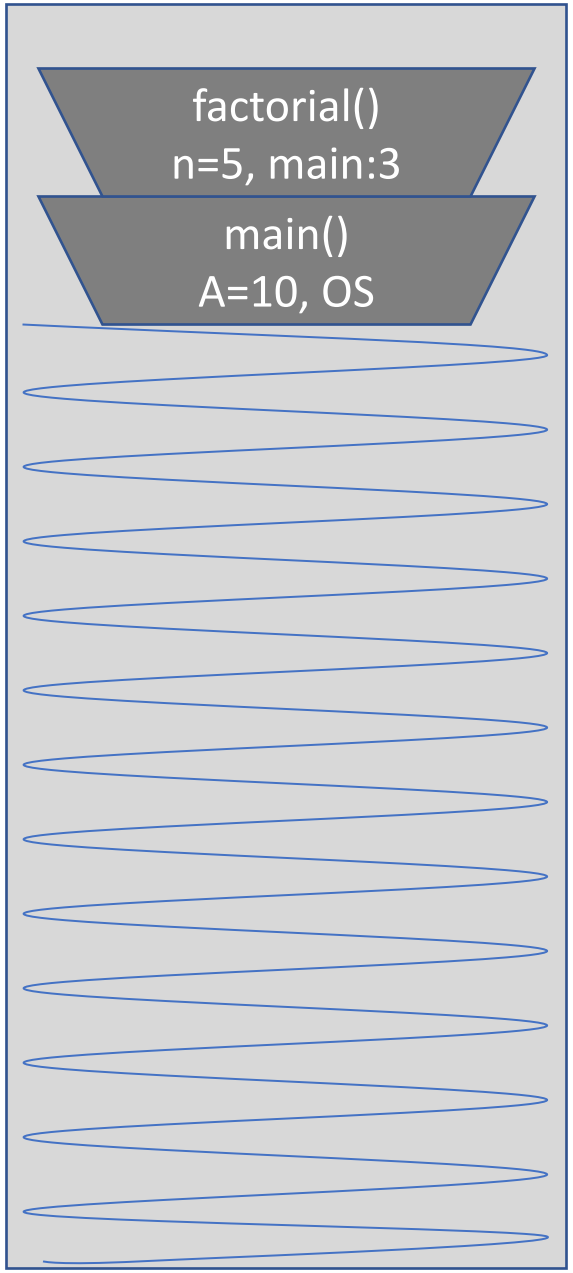
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

```
      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

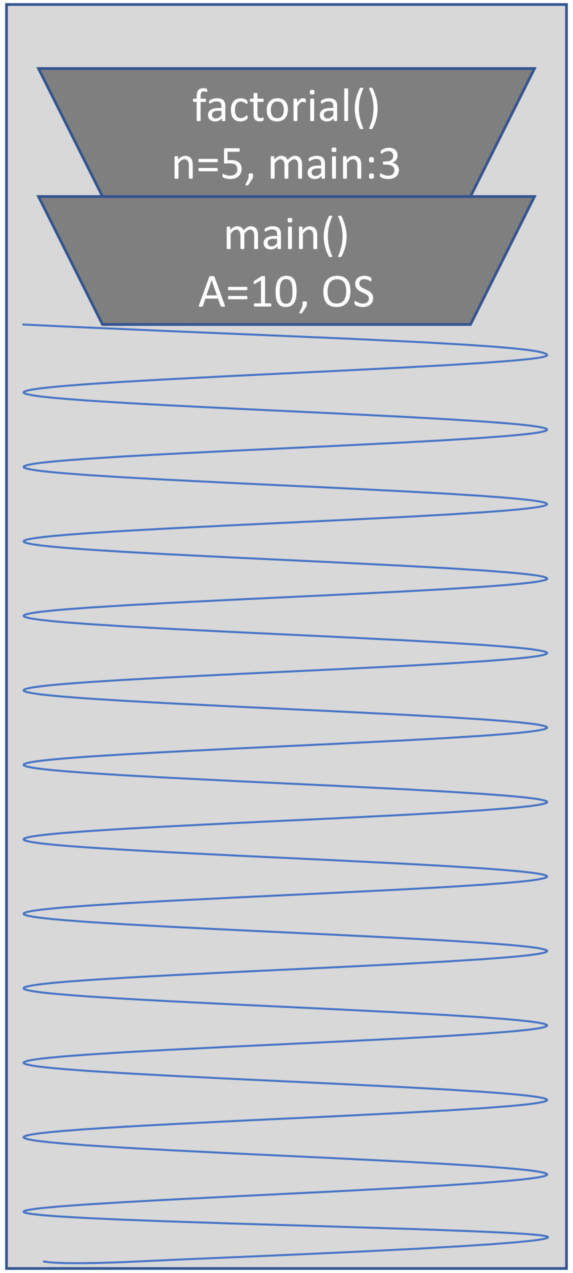
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

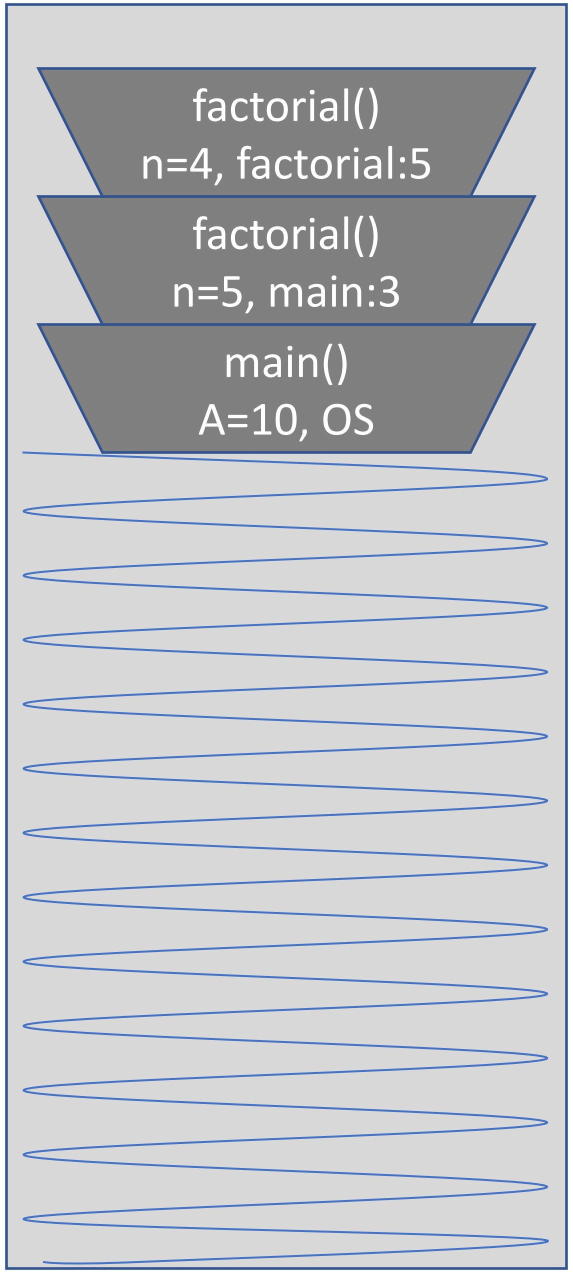
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

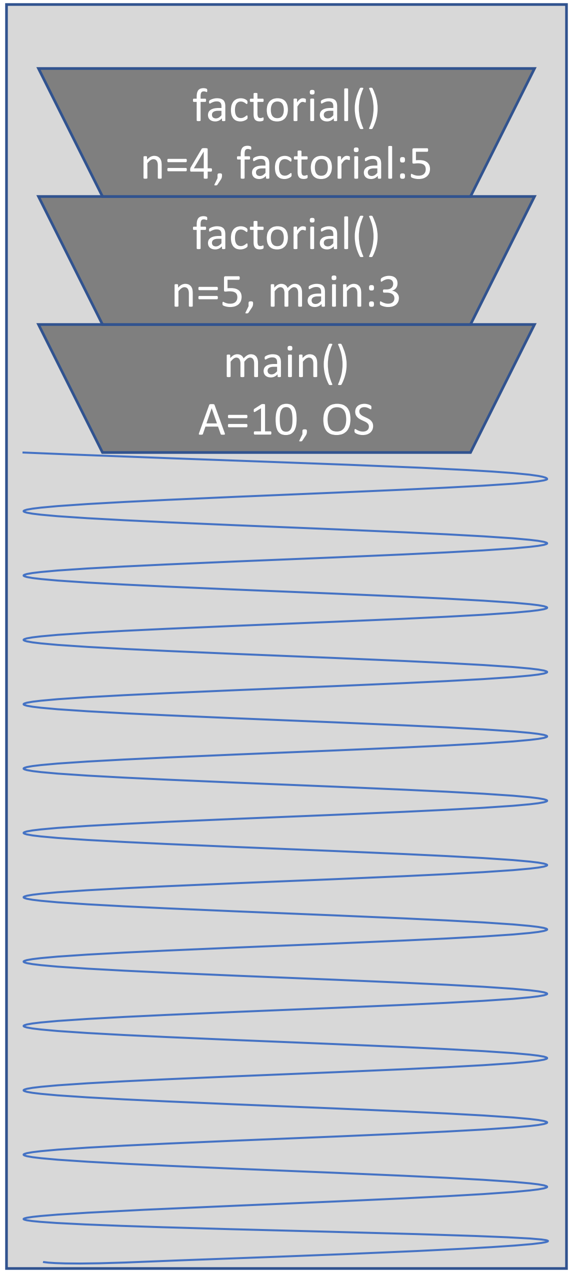
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

```
      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

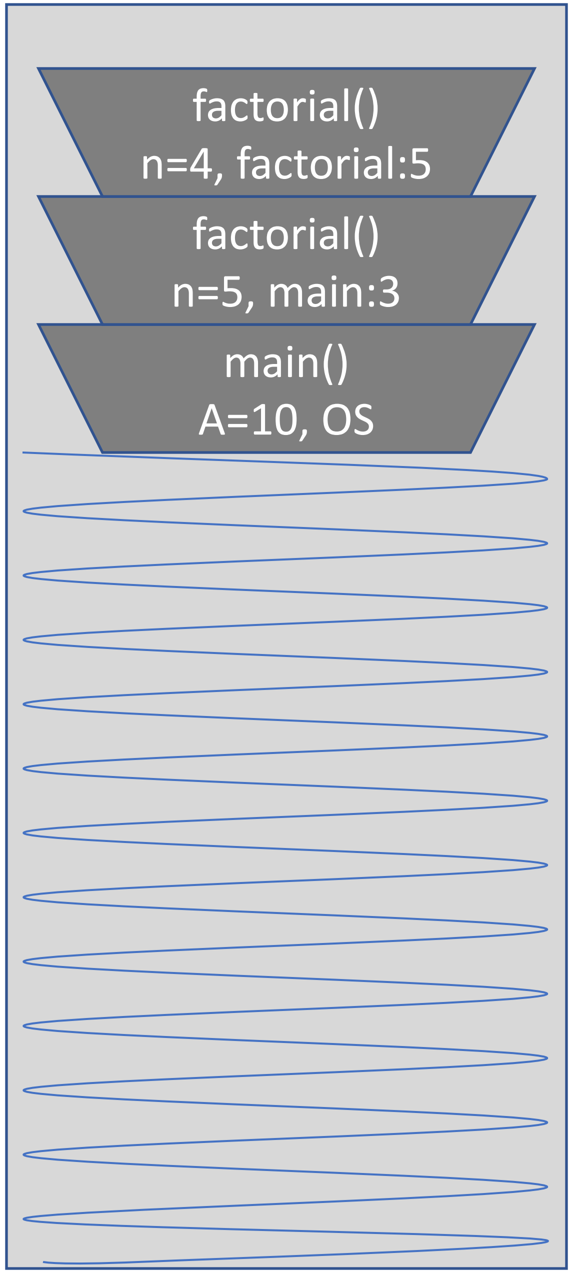
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

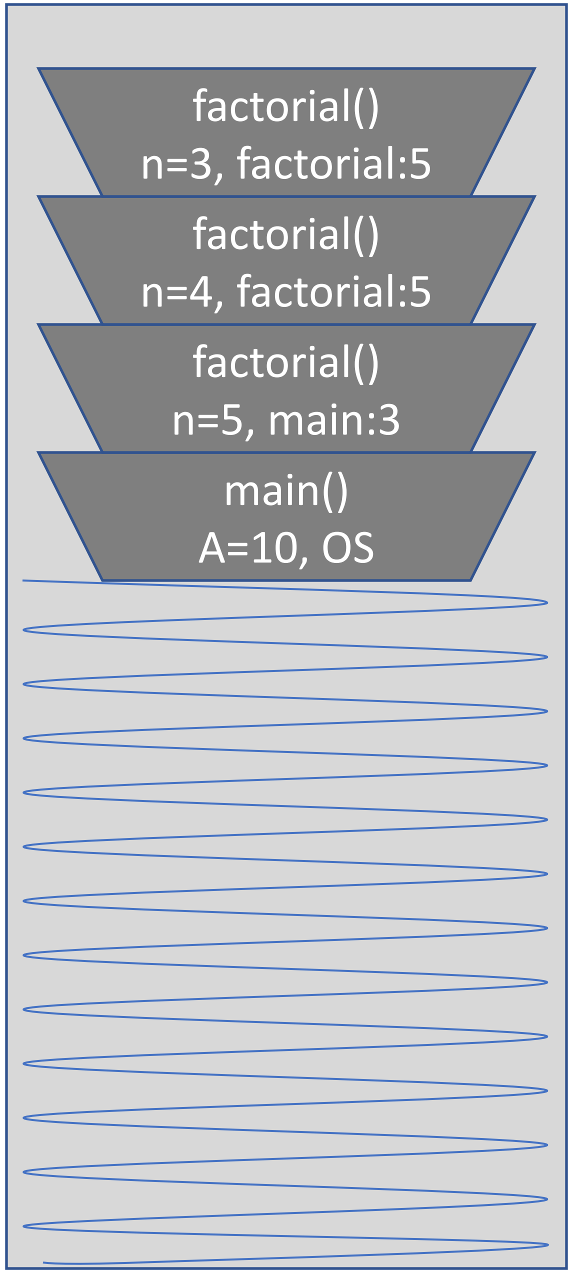
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

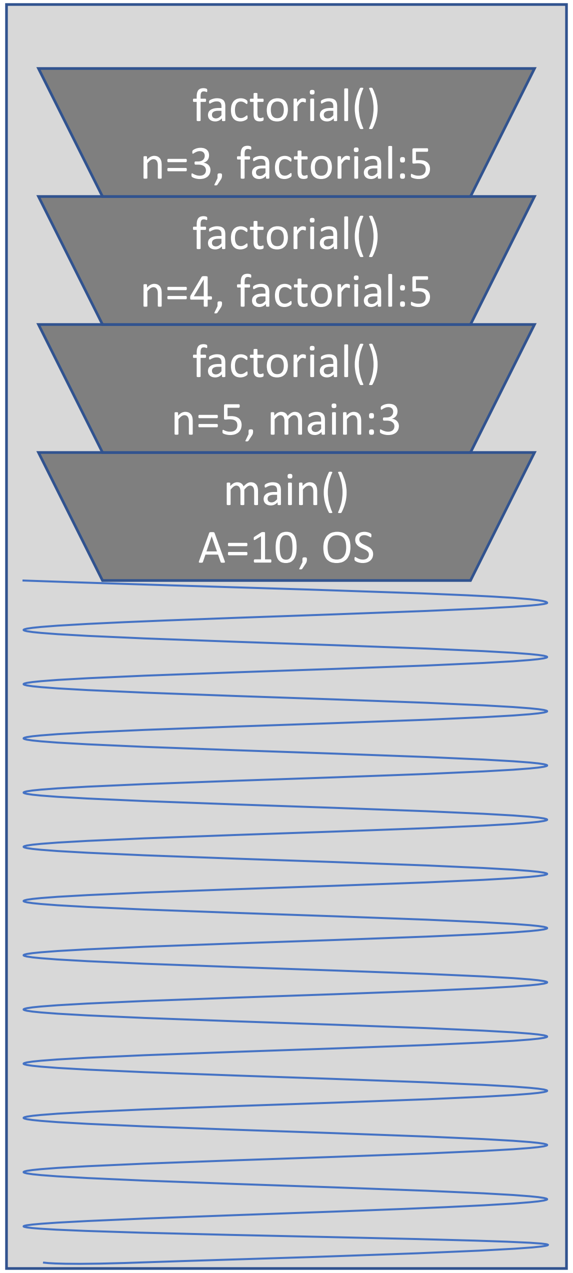
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

```
      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

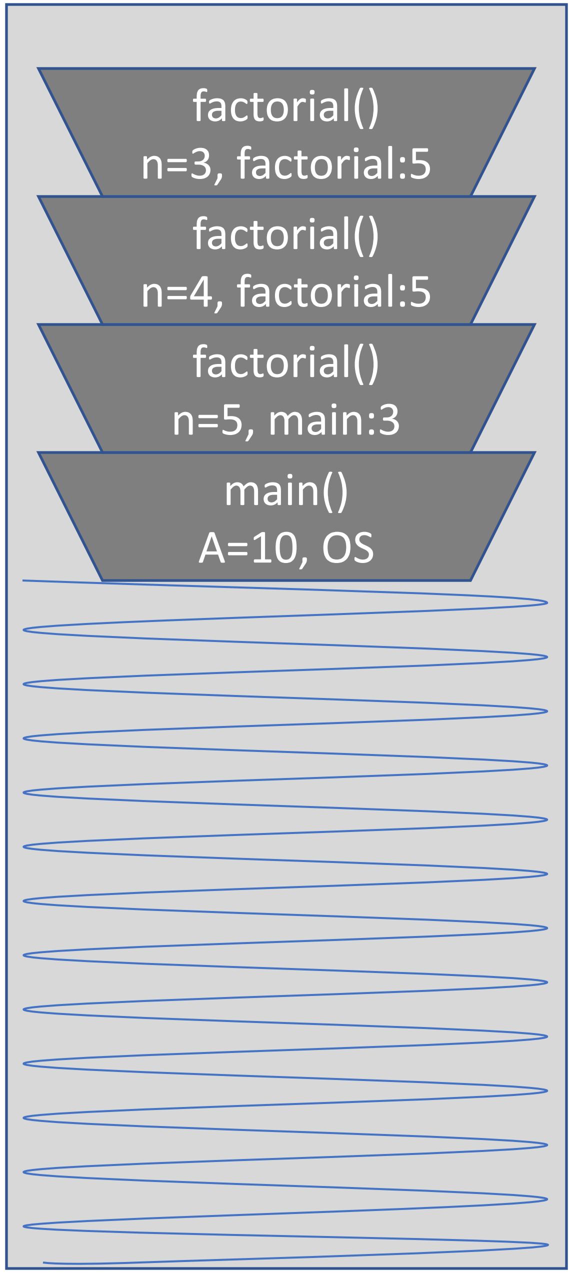
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

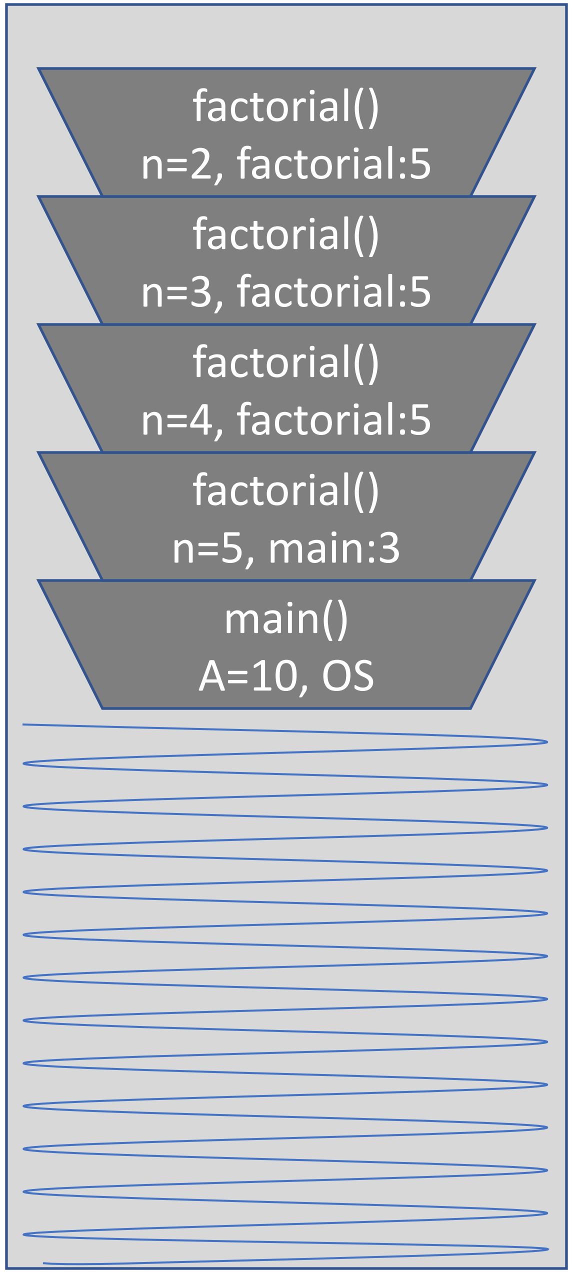
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

```
→     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

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

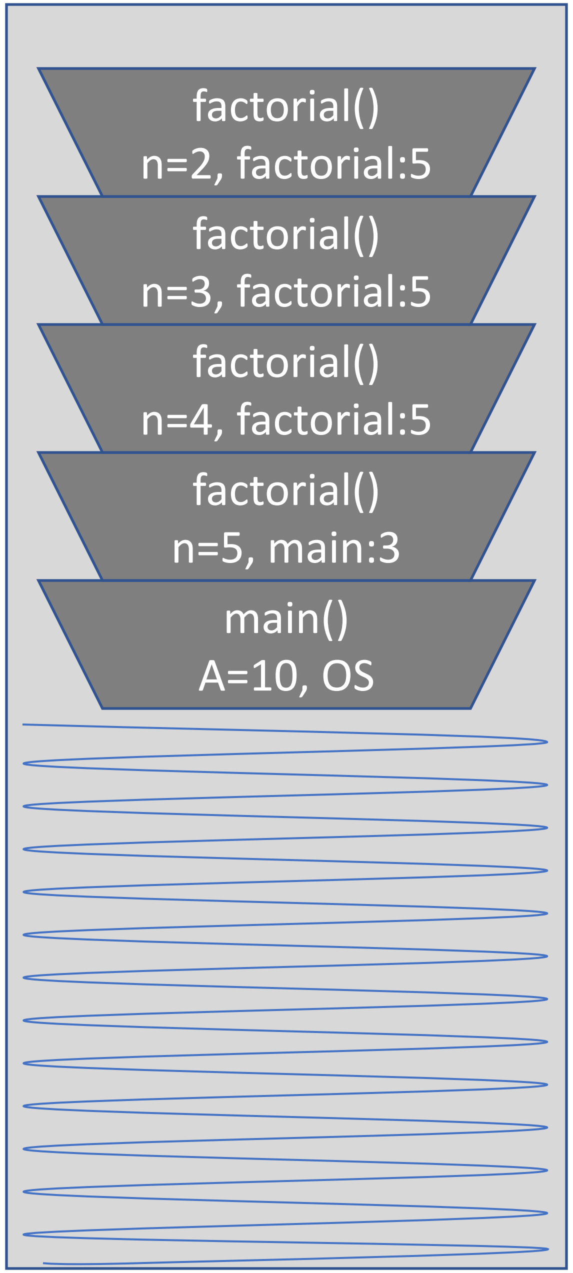
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

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



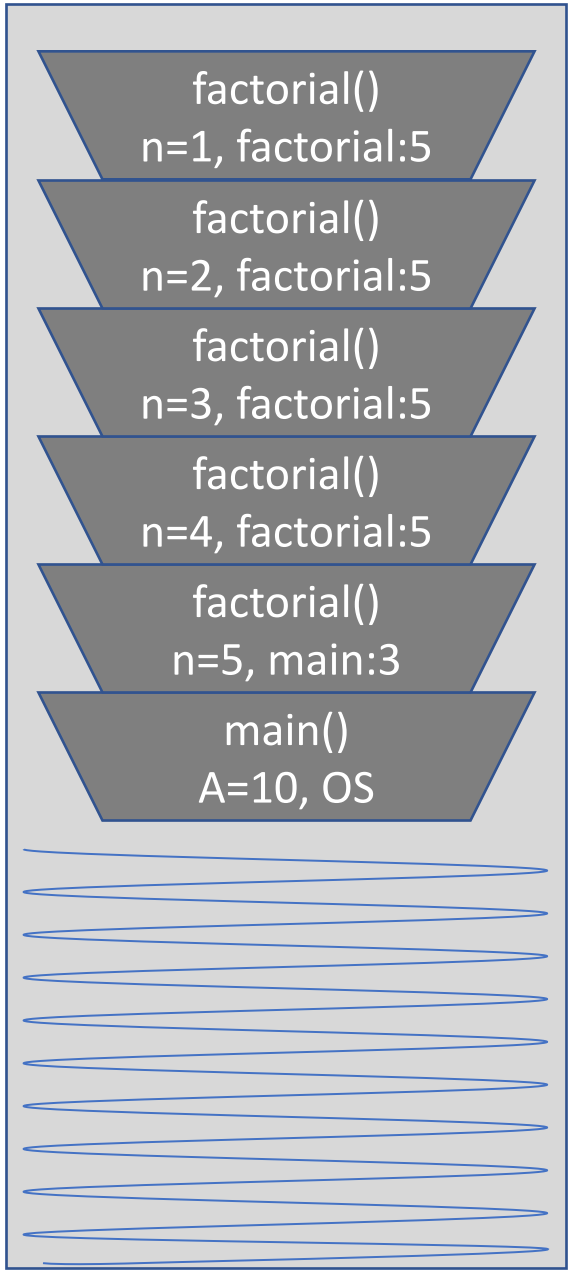
Compiled Code

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

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

Executing Function

```
int factorial(int n=2) {
  if (n == 1) {
    return 1;
  } else {
    int F = n *
factorial(n-1);
    return F;
  }
}
```



Compiled Code

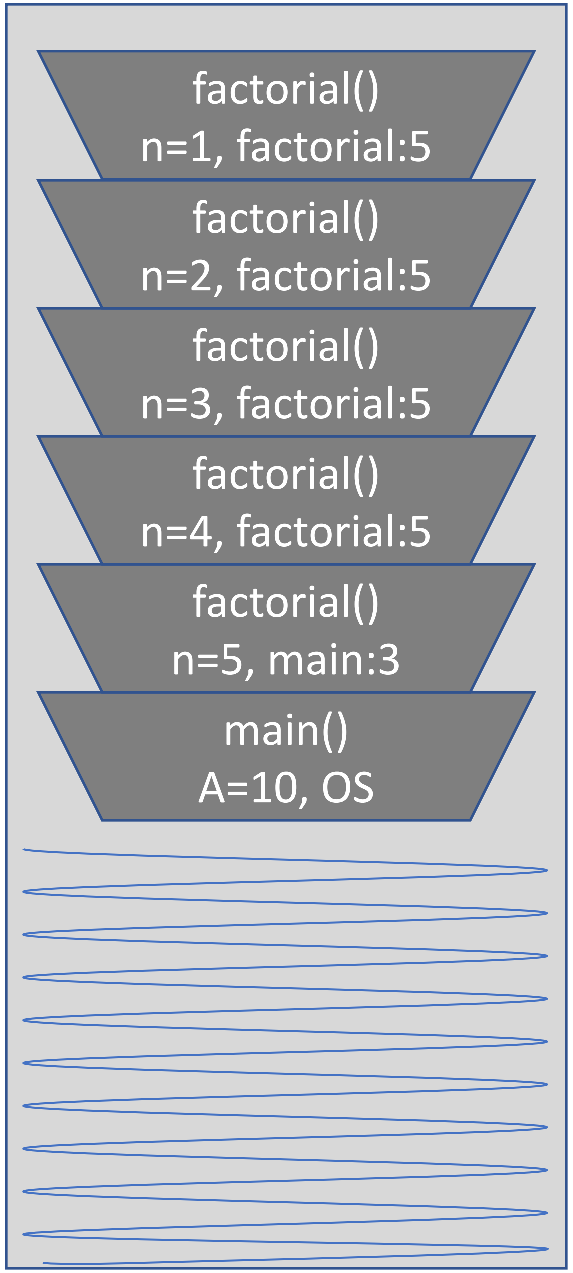
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

```
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

```
      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

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

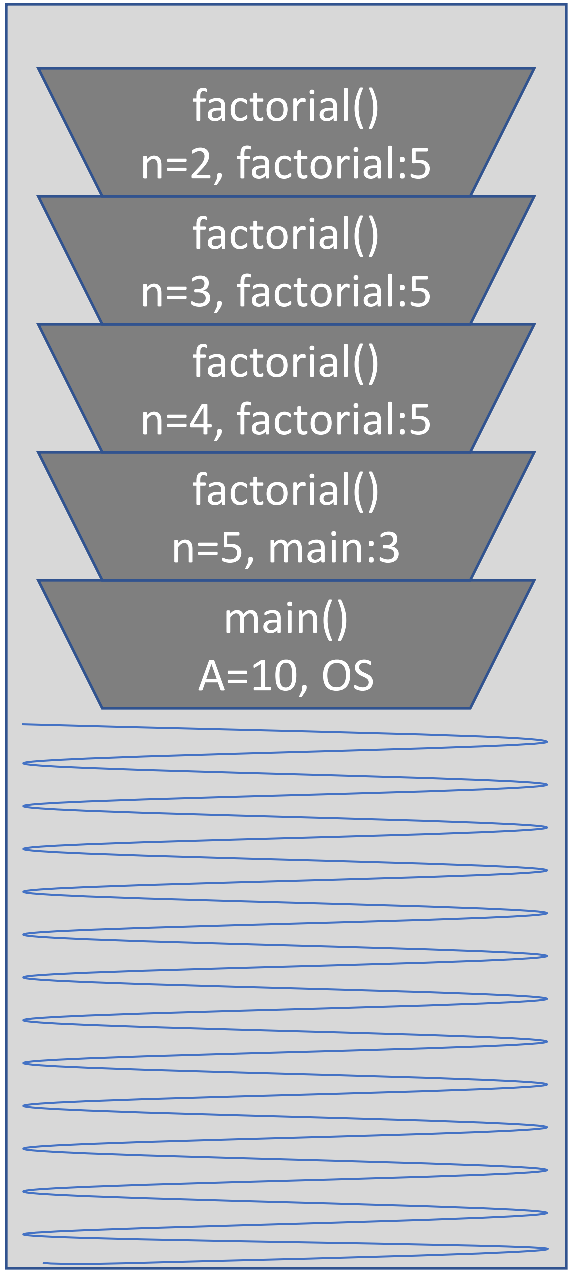
```
void main() {
  int A = 10;
  int B = factorial(5);
  System.out.println(B);
}
```

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

Executing Function

```
int factorial(int n=2) {
  if (n == 1) {
    return 1;
  } else {
    int F = n * 1;
    return F;
  }
}
```

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;
          return F;
7.      }
8.    }
```

Call Stack

factorial()
n=3, factorial:5

factorial()
n=4, factorial:5

factorial()
n=5, main:3

main()
A=10, OS

Compiled Code

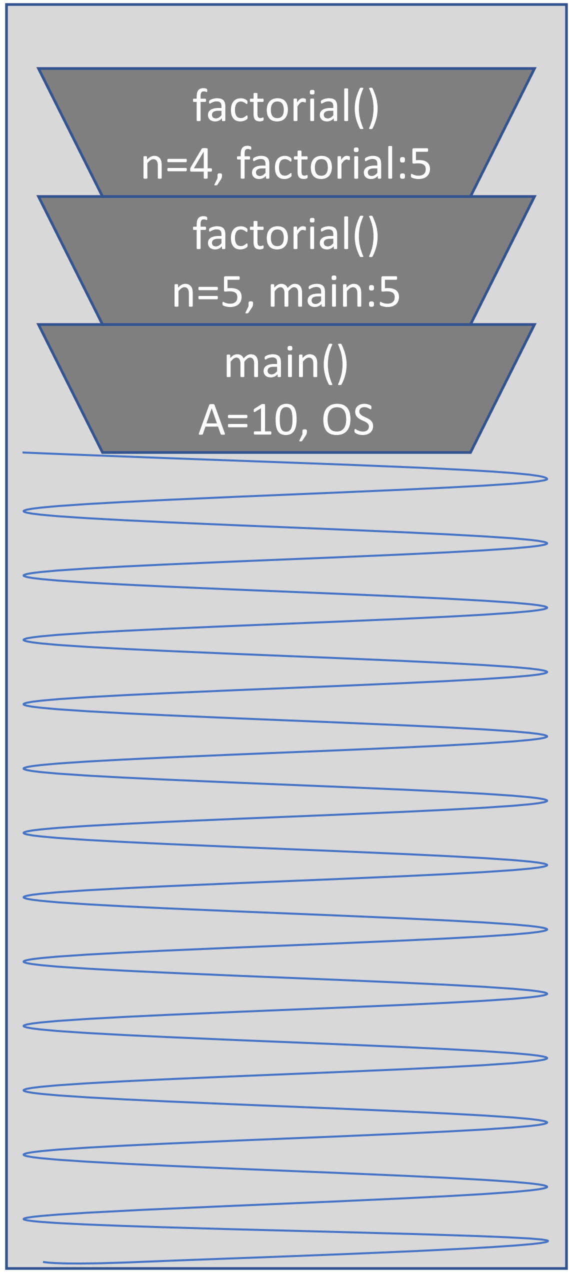
```
1.    void main() {
2.      int A = 10;
3.      int B = factorial(5);
4.      System.out.println(B);
5.    }
```

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

Call Stack



Compiled Code

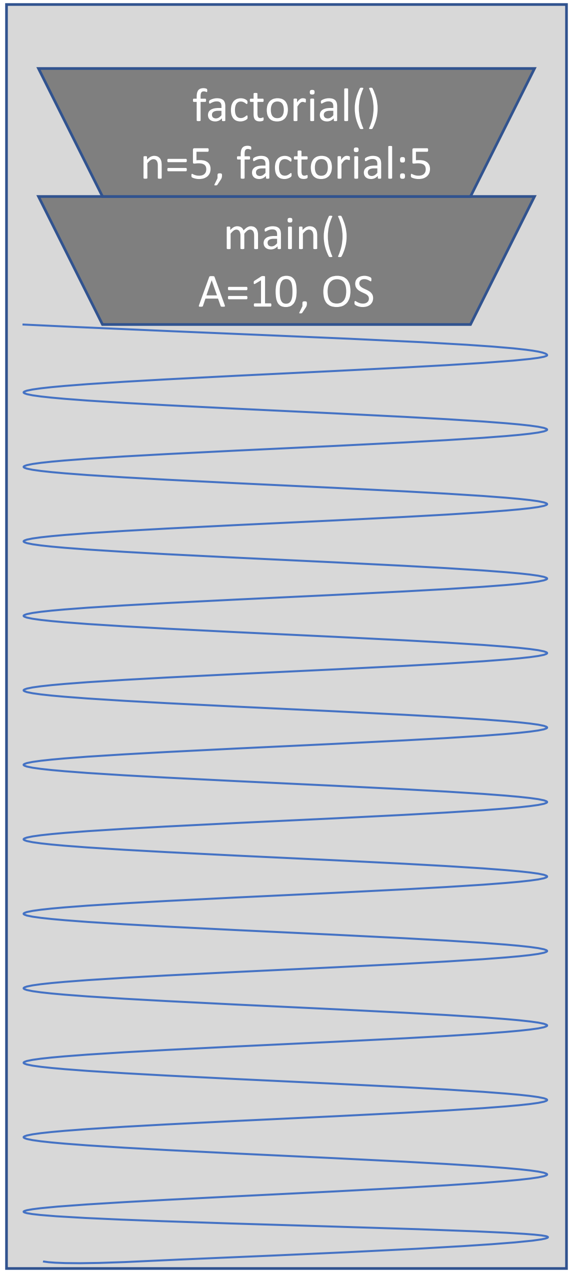
```
void main() {
  int A = 10;
  int B = factorial(5);
  System.out.println(B);
}
```

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

Executing Function

```
int factorial(int n=5) {
  if (n == 1) {
    return 1;
  } else {
    int F = n * 24;
    return F;
  }
}
```

Call Stack



Compiled Code

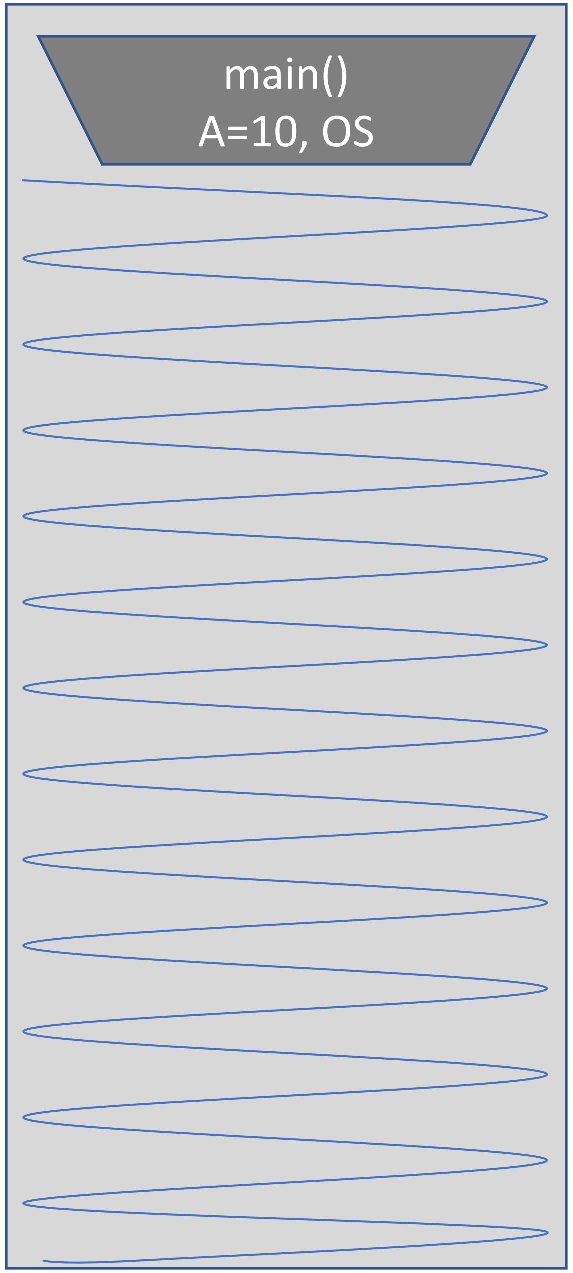
```
void main() {
  int A = 10;
  int B = factorial(5);
  System.out.println(B);
}
```

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

Executing Function

```
void main() {
  int A = 10;
  int B = 120;
  System.out.println(B);
}
```

Call Stack



Outline

- Runtime
- Recursion
- **Binary Search**

Binary Search

- efficient search in a sorted list
- can be implemented **recursively**

Search steps:

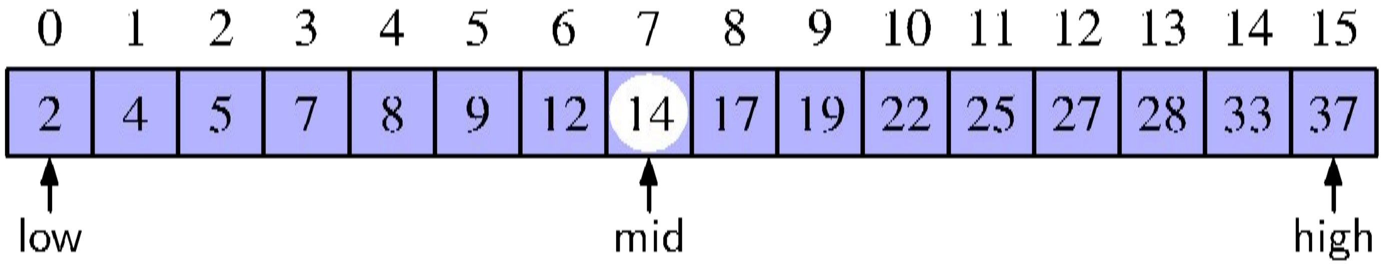
1. Calculate midpoint
2. Compare the value at the midpoint with the target value
    a. if equal:
        i. return index
    b. if target value $<$ midpoint value:
        i. **search** the left portion of the list
    c. if target value $>$ midpoint value:
        i. **search** the right portion of the list

Binary Search

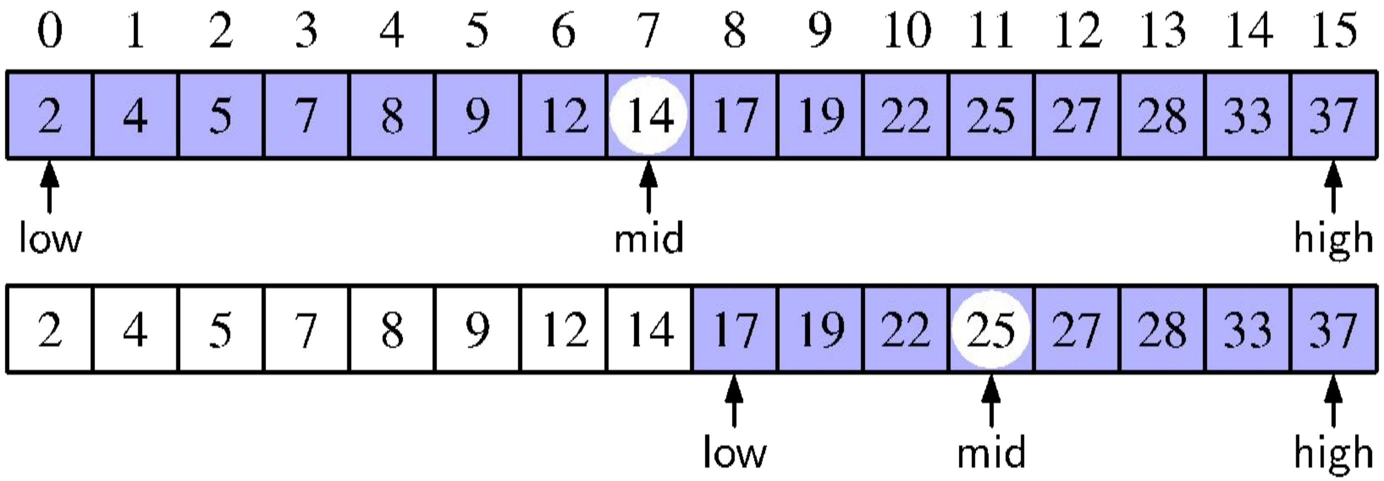
Search for an integer (22) in an ordered list

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

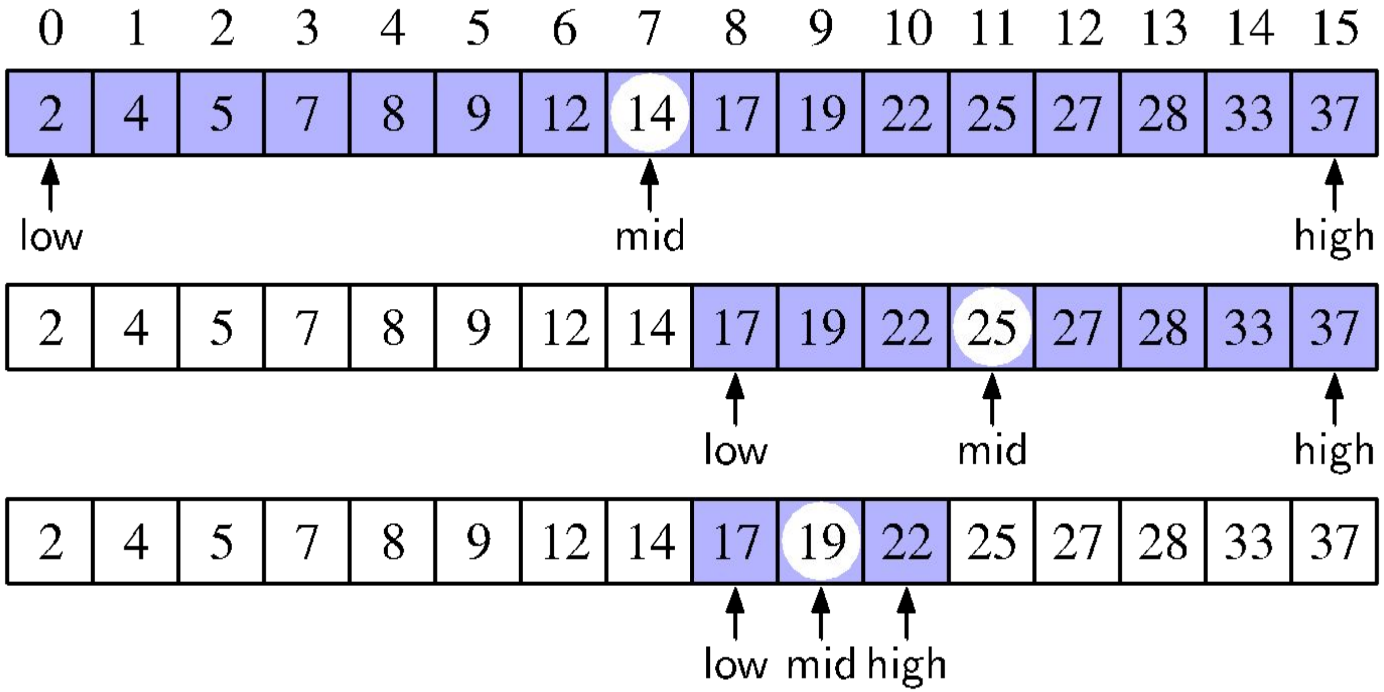
target = 22


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

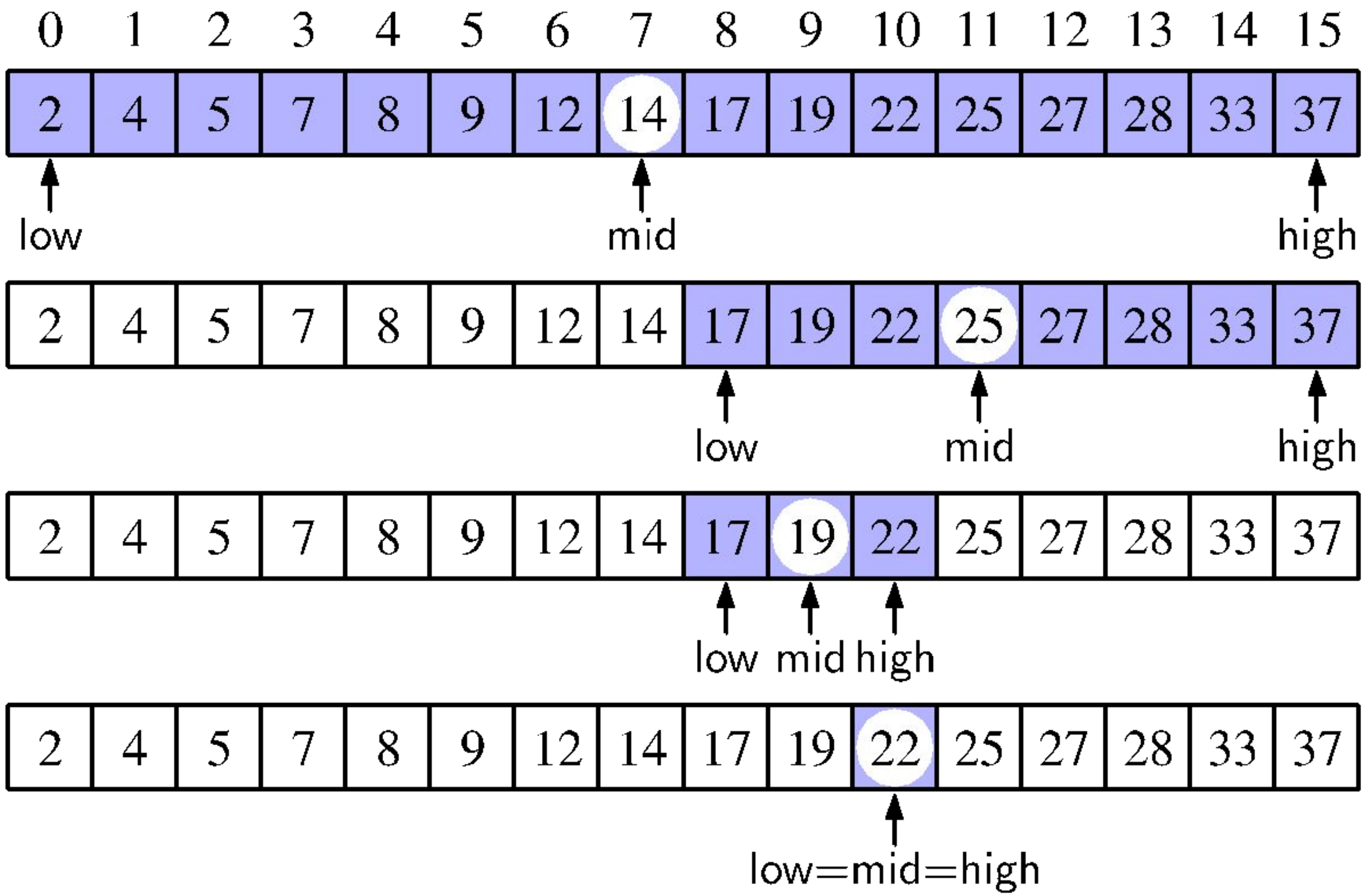
target = 22


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
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high

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
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high

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
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low mid high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low=mid=high

Binary Search Implementation

Binary Search Analysis

- Each recursive call divides the array in half

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

$O(logn)$

Comparable

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

In other words, the objects must implement `Comparable`