SAMUEL GINN
COLLEGE OF ENGINEERING

Recursion

Recursion

Recursion is a means of expressing the solution to a problem in terms of solutions to smaller instances of the same problem.

Example: How long is the line from here to the ticket window?

1 + the length of the shorter line in front of me

Recursion

Recursion is a means of expressing the solution to a problem in terms of solutions to smaller instances of the same problem.

This expression is true no matter where in the line we are.

1 + the length of the shorter line in front of me

Recursion

The ***smallest*** instance of the problem must have a known solution or be trivial to compute; that is, one that does not involve recursion.

We call this smallest instance of the problem the **base case**.

1

Recursion

Recursive solutions typically have two parts: one that describes the solution to the smallest instance of the problem, and one that describes the general instance of the problem.

Example: How long is the line from here to the ticket window?

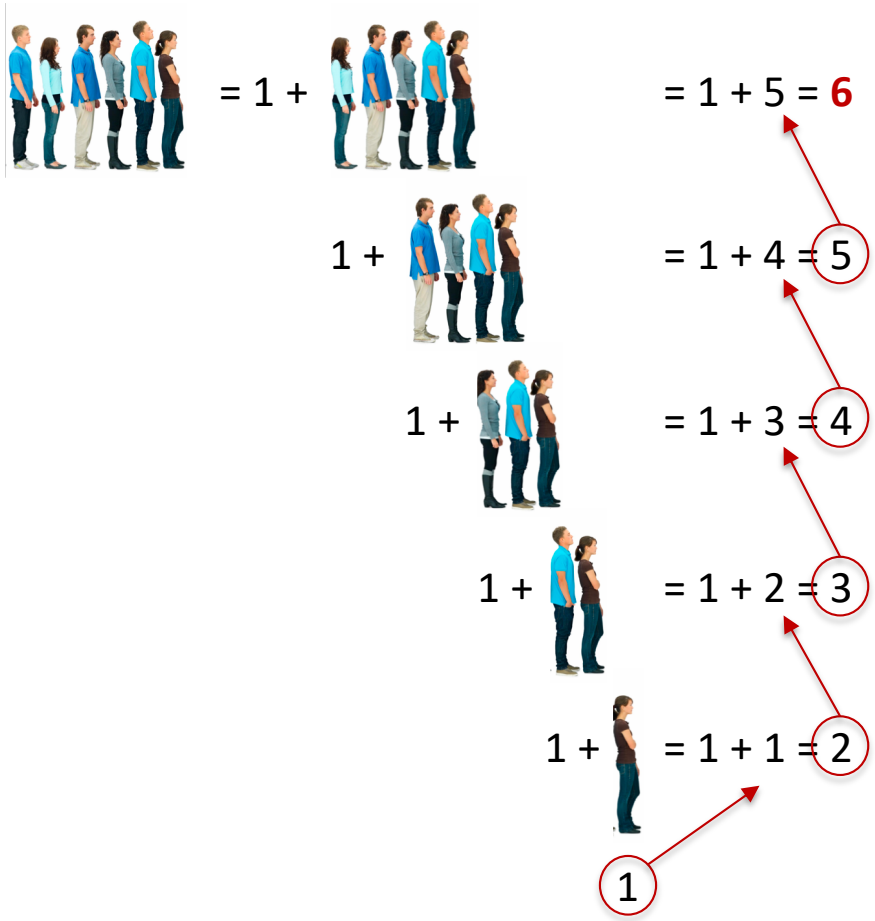
If there is no one in front of me, the length of the line is 1.

← **Base case**

Otherwise, the length of the line is 1 plus the length of the smaller line in front of me.

← **Reduction Step (recursive case)**

Recursive evaluation



There are 6 people in line.

Recursion v. iteration

How else might we solve the line length problem? An intuitive approach might be to walk down the line, counting people as we go.

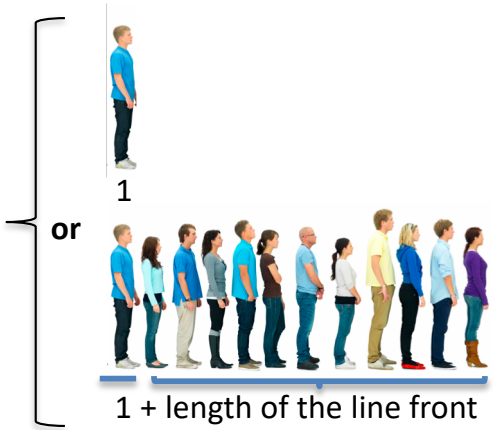
This is an **iterative** approach, and we could express this using the linear scan strategy.

```
initialize length to 1
for (each person in front of me in line) {
   add 1 to length
}
length stores the number of people in line
```

1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 = 12

Contrast this with our **recursive** approach:

```
if (there is no one in front of me) {
   length is 1
} else {
   length is 1 + the length of the line in
                                in front of me
}
```

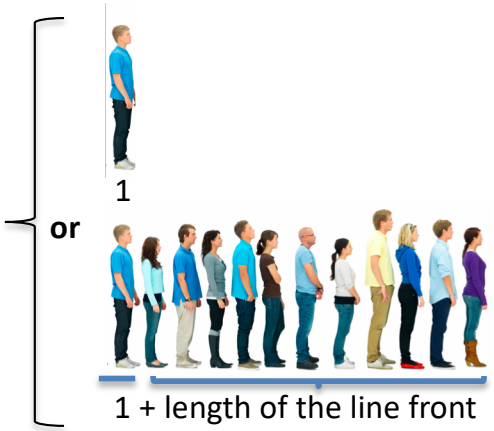


Expressing recursion in Java

Self-reference is at the heart of recursion, and expressing recursive solutions in Java relies on a method being able to call itself.

This recursive strategy:

```
if (there is no one in front of me) {
   length is 1
} else {
   length is 1 + the length of the line in
                              in front of me
}
```

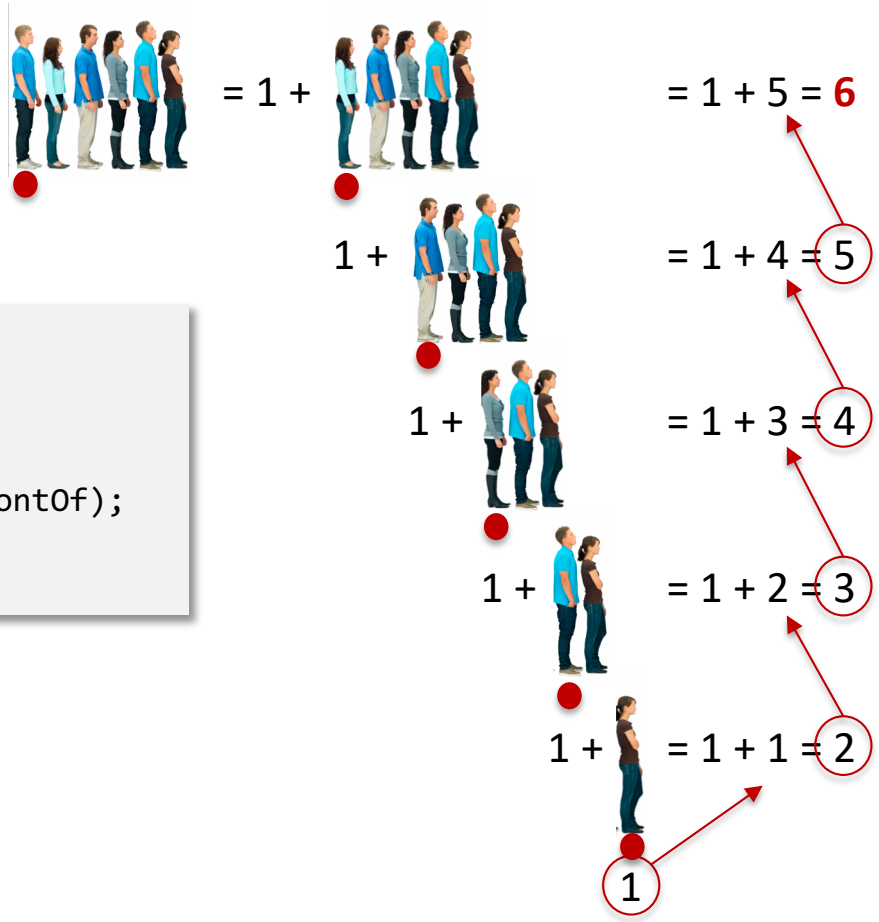


Would be expressed something like this:

```
public int length(Person person) {
      if (person.inFrontOf == null) {
            return 1;
      } else {
            return 1 + length(person.inFrontOf);
      }
}
```

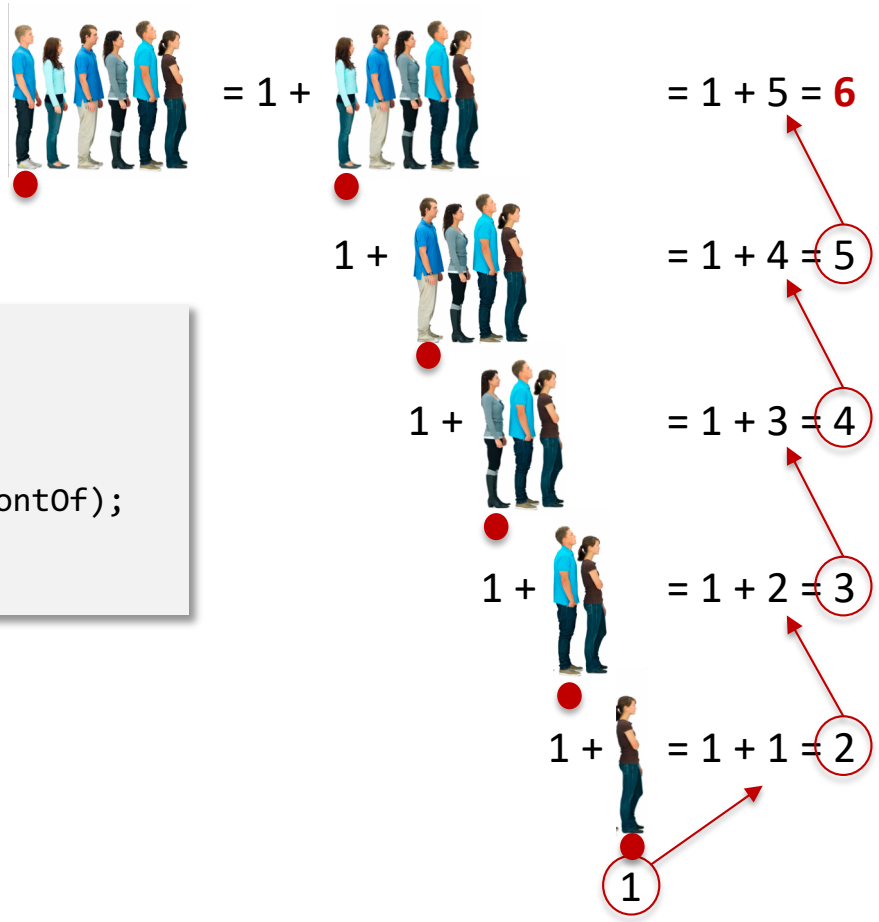
Expressing recursion in Java

```
public int length(Person person) {
      if (person.inFrontOf == null) {
            return 1;
      } else {
            return 1 + length(person.inFrontOf);
      }
}
```



Expressing recursion in Java

```
public int length(Person person) {
      if (person.inFrontOf == null) {
            return 1;
      } else {
            return 1 + length(person.inFrontOf);
      }
}
```



Factorial

Factorial

Definition: The factorial of a positive integer n, denoted n!, is the product of all the positive integers less than or equal to n.

$$n! = \prod_{i=1}^{n} i$$

Example: 5! = 5 x 4 x 3 x 2 x 1 = 120

Iterative implementation:

```
public int factorial(int n) {
    int fact = n;
    for (int i = n - 1; i > 0; i--) {
        fact = fact * i;
    }
    return fact;
}
```

Factorial

Remember that recursion is a means of expressing the solution to a problem in terms of solutions to smaller instances of the same problem, where the *smallest* instance of the problem must have a known solution or be trivial to compute.

So, to express the computation of factorial recursively, we need two things: (1) the solution of the smallest instance of the factorial problem (**base case**), and (2) a statement of how to compute factorial in terms of smaller factorial problem (**reduction step**).

The smallest factorial problem is 1!, since we defined factorial only on the positive integers and 1 is the smallest positive integer. We also know that 1! = 1 since the product of all the positive integers less than or equal to 1 is 1.

Now we can express the **base case** of our recursive computation of factorial:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    . . .
}
```

Factorial

To express factorial in terms of smaller factorial problems, let's revisit our expansion of 5! below.

We want to identify smaller sub-solutions in the expansion, and think of how to specify a reduction or decomposition.

5! = 5 x 4 x 3 x 2 x 1 = 120

This is 4!

5! = 5 x 4!

5! = 5 x 4 x 3!

5! = 5 x 4 x 3 x 2!

5! = 5 x 4 x 3 x 2 x 1!

5! = 5 x 4 x 3 x 2 x 1 = 120

$$n! = \prod_{i=1}^{n} i \;\; = n \times \prod_{i=1}^{n-1} i$$

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases}$$

Factorial

A recursive definition of factorial:

$$n! = \begin{cases} 1 & \text{if } n = 1 \leftarrow \textbf{Base case} \\ n \times (n-1)! & \text{if } n > 1 \leftarrow \textbf{Reduction step} \end{cases}$$

The **base case** states the solution to the smallest instance of the problem.

The **reduction step** reduces all other problems to the base case.

Factorial

A recursive computation of factorial:

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

Base case

Reduction step

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases}$$

Factorial

A recursive computation of factorial:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
factorial(5)
5 * factorial(4)
    4 * factorial(3)
        3 * factorial(2)
            2 * factorial(1)
                1
            2 * 1
        3 * 2
    4 * 6
5 * 24 = 120
```

Factorial

Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF

Factorial

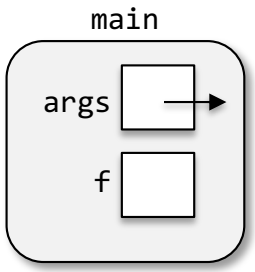
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

`0x00000000` ⟶ stack memory `0x7FFFFFFF`



Factorial

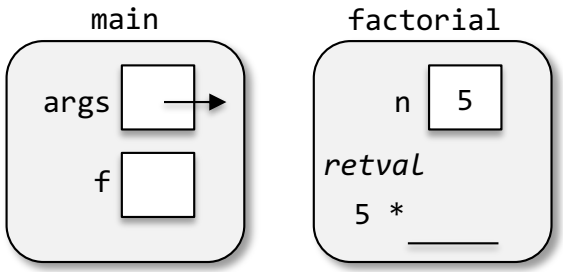
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



Factorial

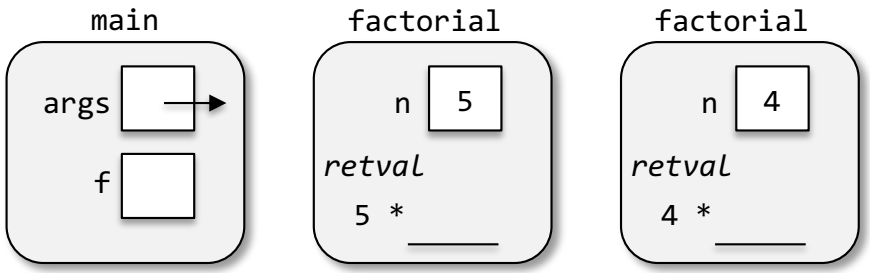
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



Factorial

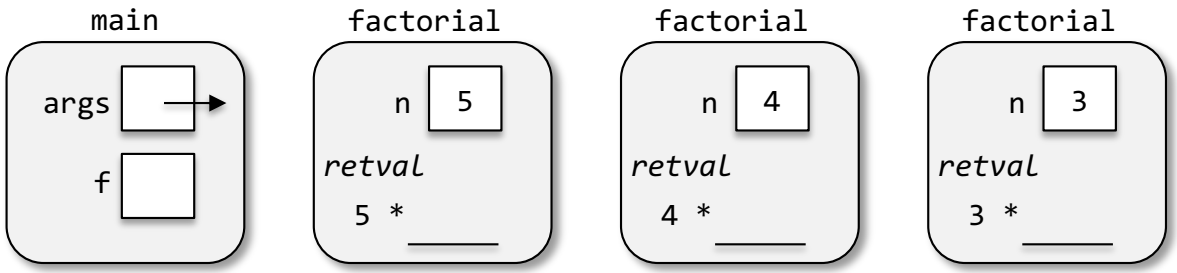
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



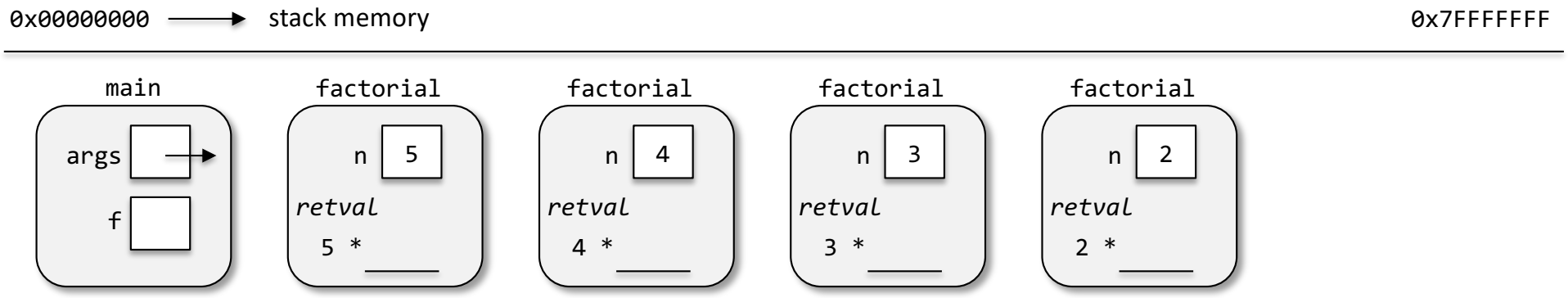
Factorial

Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:



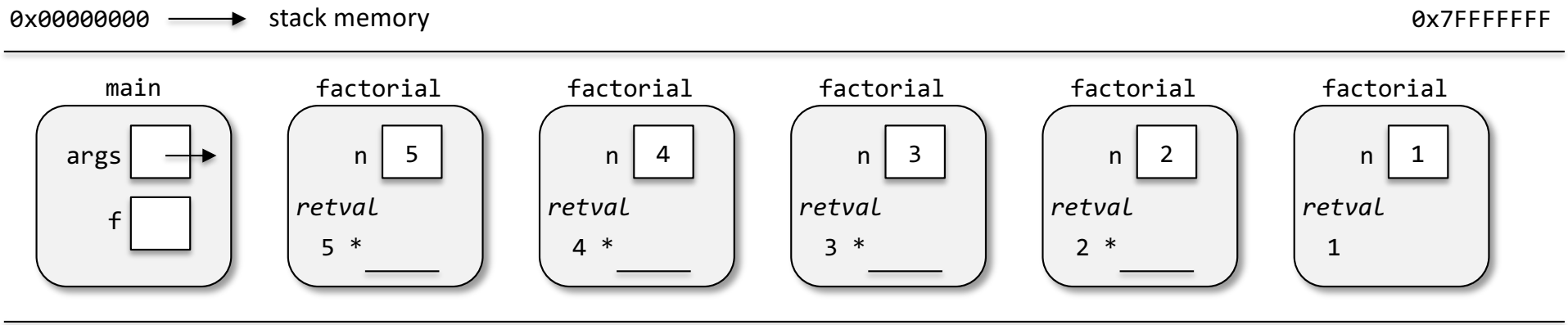
Factorial

Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:



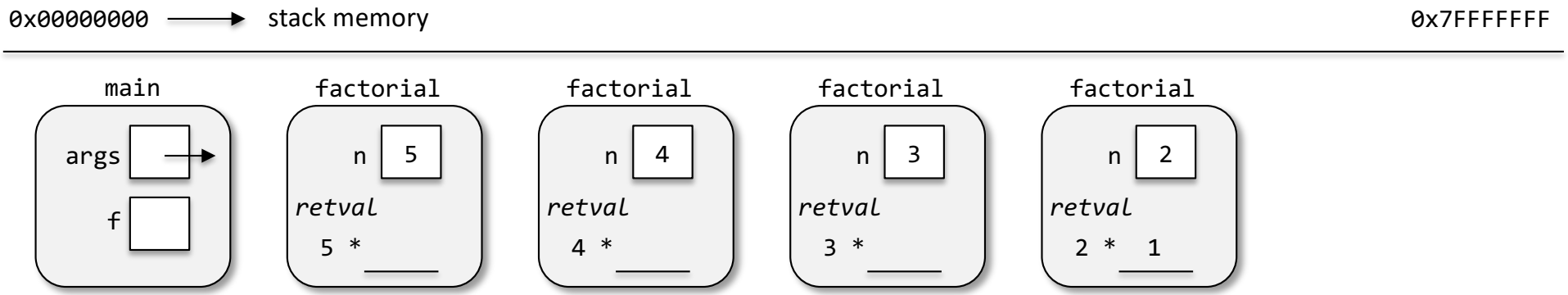
Factorial

Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:



Factorial

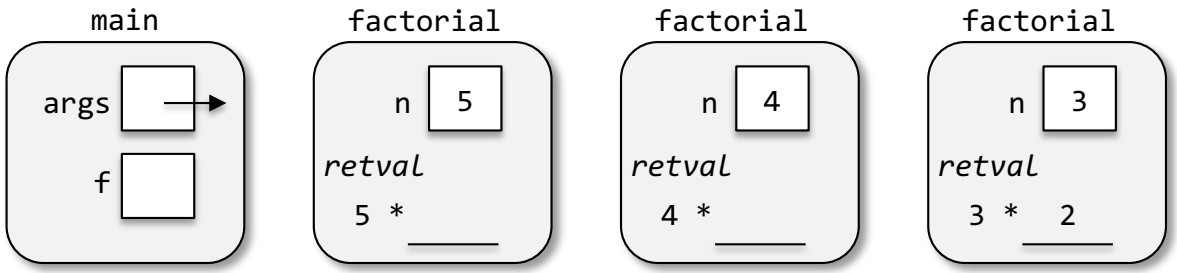
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



Factorial

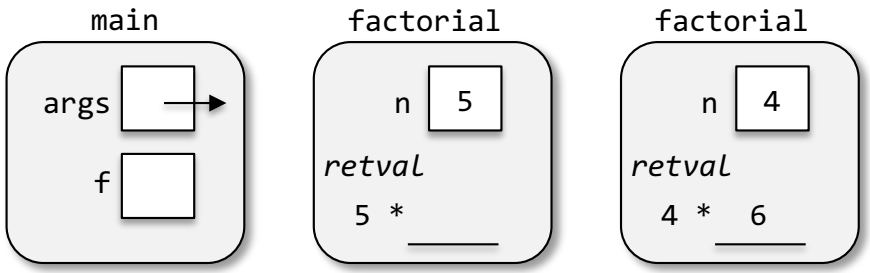
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



Factorial

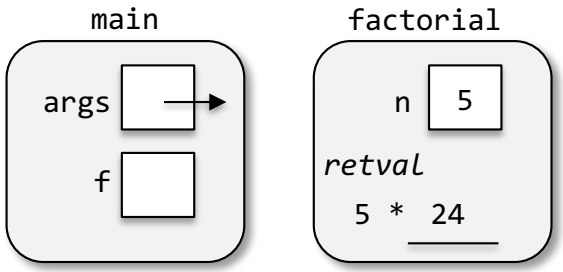
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



Factorial

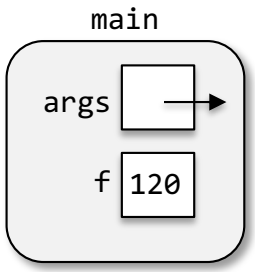
Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF



Factorial

Tracing the execution of this method on the call stack:

```
public int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n – 1);
}
```

```
public static void main(String[] args) {
    int f = factorial(5);
}
```

Internal memory:

0x00000000 ⟶ stack memory 0x7FFFFFFF

Tower of Hanoi

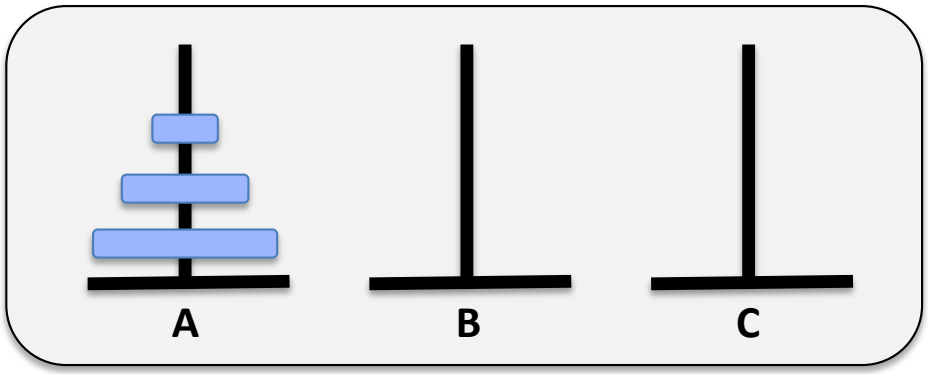
The power of recursive thinking

You've been hired to develop control software for a robotic lift in a warehouse. The robotic lift must move stacks of widgets from the incoming shipping bay to a long-term storage bay. The widgets are stacked one on top of another, from largest on the bottom to smallest on the top. The robotic lift can only move one widget at a time due to weight, and can never stack a larger widget on top of a smaller widget. There is one temporary storage bay that the robotic lift can use while moving the widgets from the shipping bay to the long-term storage bay. At all times each storage bay can hold at most one stack of widgets, and the stacks must be arranged from largest on bottom to smallest on top. The transfer of a single widget from the stack in one bay to the stack in another bay (shipping, temporary, or long-term) is considered a "move." You must write the control software so that the robotic lift makes the minimum number of widget moves possible.

Tower of Hanoi

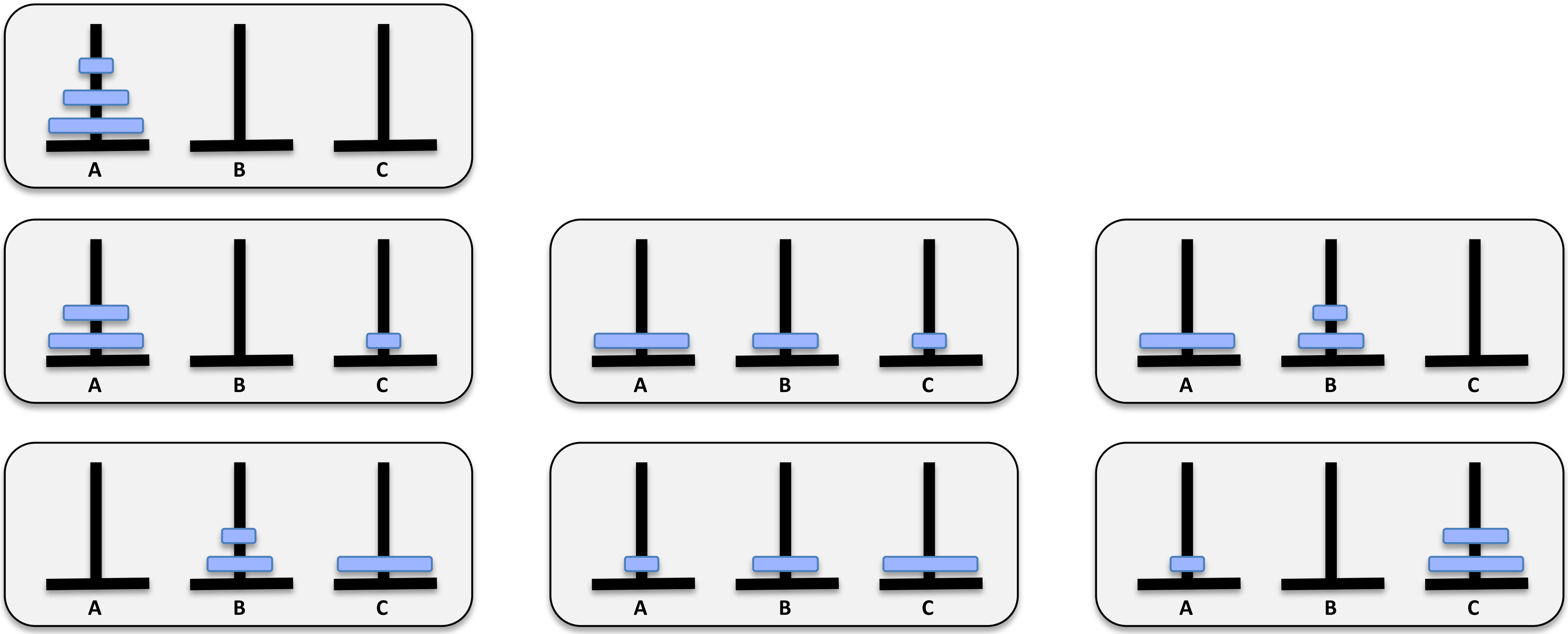
Given three pegs (A, B, and C) and N disks, with the initial configuration of all N disks stacked from largest to smallest on peg A, move all N disks from peg A to peg C while obeying the following rules.

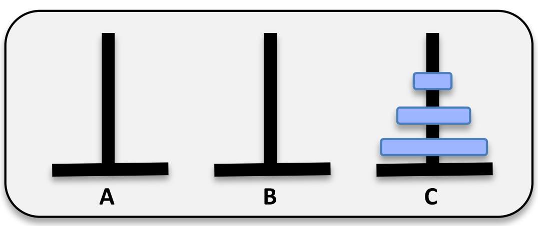
1. Only the top disk on a stack can be moved.
2. Only one disk can be moved at a time.
3. No disk can be placed on top of a smaller disk.



Write an algorithm that makes the fewest number of moves for N disks.

Tower of Hanoi

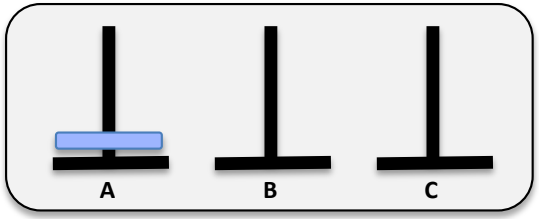




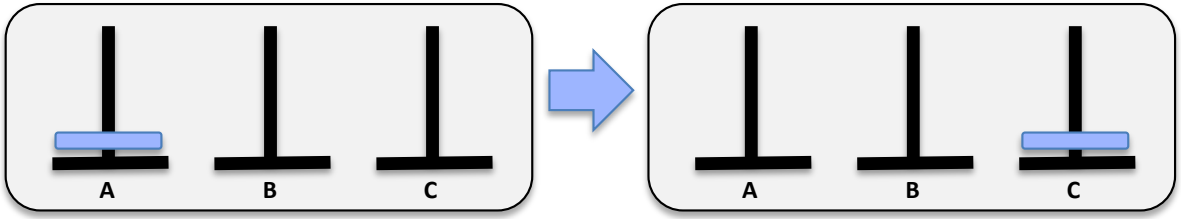
($2^N - 1$ is the fewest number of moves possible with N disks.)

Tower of Hanoi

Base case: One disk.

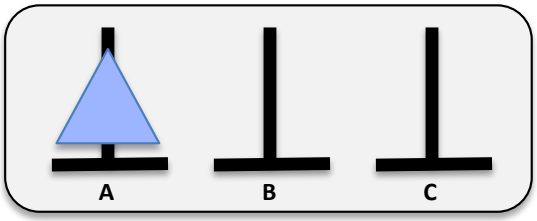


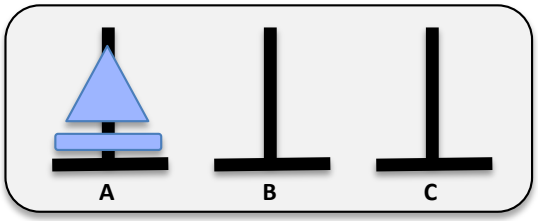
Solution: Move the disk from A to C.



Reduction step: N disks.

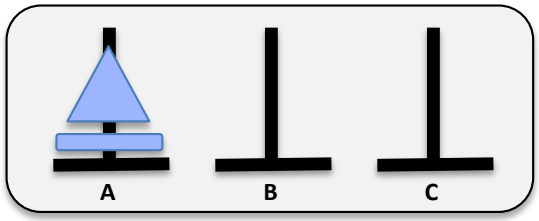
The key insight is in how we view these N disks.



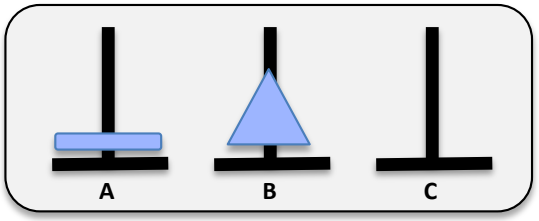


Tower of Hanoi

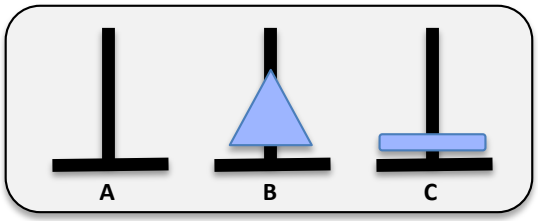
Reduction step: N disks.



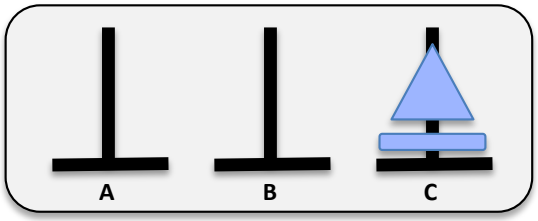
Solution: (1) Recursively move N-1 disks from A to B.



(2) Move 1 disk from A to C.



(3) Recursively move N-1 disks from B to C.

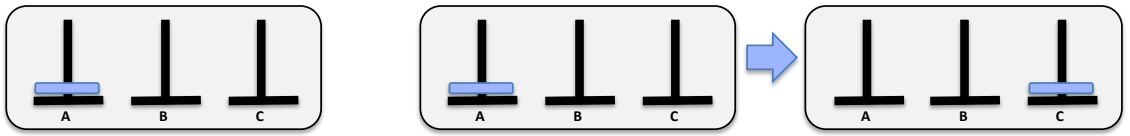


Tower of Hanoi

```
/** Recursive solution to Tower of Hanoi. Prints optimal sequence of moves. */
public static void hanoi(int n, String startPeg, String endPeg, String tempPeg) {
      if (n == 1) {
            System.out.println("Move one disk from " + startPeg + " to " + endPeg + ". ");
      } else {
            hanoi(n - 1, startPeg, tempPeg, endPeg);
            System.out.println("Move one disk from " + startPeg + " to " + endPeg + ". ");
            hanoi(n - 1, tempPeg, endPeg, startPeg);
      }
}
```

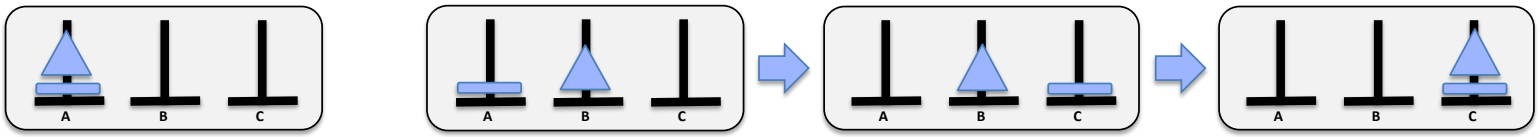
Tower of Hanoi

Base Case:




```
/** Recursive solution to Tower of Hanoi. Prints optimal sequence of moves. */
public static void hanoi(int n, String startPeg, String endPeg, String tempPeg) {
    if (n == 1) {
        System.out.println("Move one disk from " + startPeg + " to " + endPeg + ". ");
    } else {
        hanoi(n - 1, startPeg, tempPeg, endPeg);
        System.out.println("Move one disk from " + startPeg + " to " + endPeg + ". ");
        hanoi(n - 1, tempPeg, endPeg, startPeg);
    }
}
```

Reduction Step:

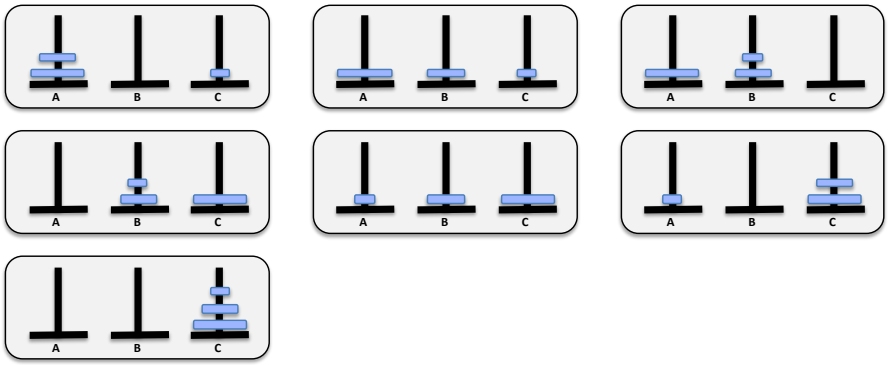


Tower of Hanoi

```
/** Recursive solution to Tower of Hanoi. Prints optimal sequence of moves. */
public static void hanoi(int n, String startPeg, String endPeg, String tempPeg) {
      if (n == 1) {
            System.out.println("Move one disk from " + startPeg + " to " + endPeg + ". ");
      } else {
            hanoi(n - 1, startPeg, tempPeg, endPeg);
            System.out.println("Move one disk from " + startPeg + " to " + endPeg + ". ");
            hanoi(n - 1, tempPeg, endPeg, startPeg);
      }
}
```

Output from: `hanoi(3, "A", "C", "B");`

```
Move one disk from A to C.
Move one disk from A to B.
Move one disk from C to B.
Move one disk from A to C.
Move one disk from B to A.
Move one disk from B to C.
Move one disk from A to C.
```

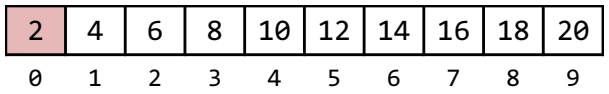


Array search

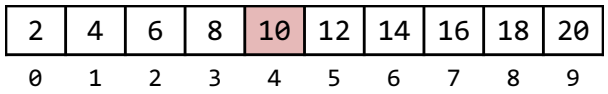
Array search

Given an array, return the index of a target value or -1 if the target value is not present.

Let's think back to our iterative expressions of **linear search** and **binary search**:




```
public static int linearSearch(int[] a, int target) {
   for (int i = 0; i < a.length; i++) {
      if (a[i] == target) {
         return i;
      }
   }
   return -1;
}
```



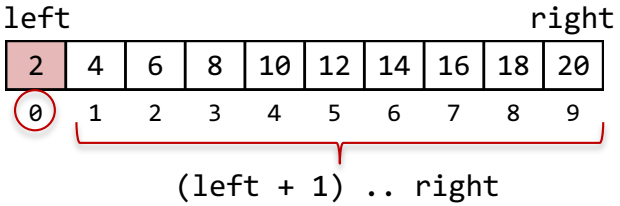

```
public int binarySearch(int[] a, int target) {
    int left = 0;
    int right = a.length - 1;
    while (left <= right) {
        int middle = left + (right - left) / 2;
        if (a[middle] == target) {
            return middle;
        }
        if (a[middle] > target)
            right = middle - 1;
         } else {
            left = middle + 1;
         }
     }
     return -1;
}
```

Array search

To develop recursive expressions of linear search and binary search, let's begin by making the following observation about both algorithms:

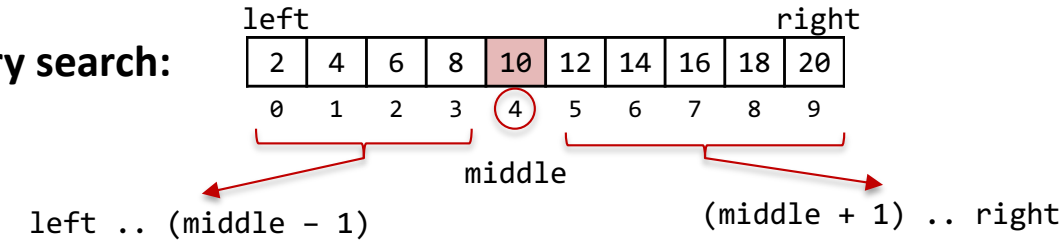
In the general case, the result of the search is either the index of the current element (if the current element is equal to target) or it is the result of searching a smaller range of elements that remain.

Linear search:



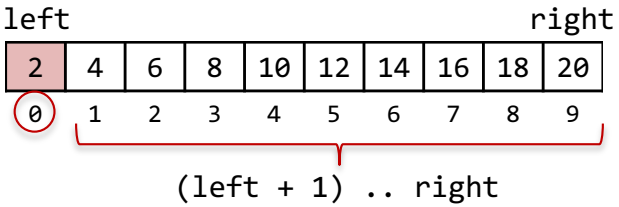
Use index variables to identify the current index and the smaller range of elements that remain.

Binary search:



Array search

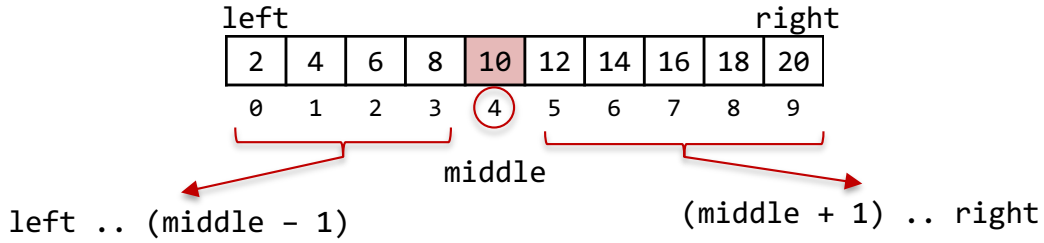
Now we can express the reduction step for linear search:




```
public static int linearSearch(int[] a, int target, int left, int right) {
    // base case goes here ...

    if (a[left] == target) {
        return left;
    }
    return linearSearch(a, target, left + 1, right);
  }
```

Array search

And for binary search:



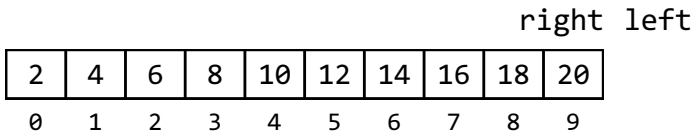

```
public int binarySearch(int[] a, int target, int left, int right) {
    // base case goes here ...

    int middle = left + (right - left) / 2;
    if (a[middle] == target) {
        return middle;
    }
    if (a[middle] > target) {
        return binarySearch(a, target, left, middle - 1);
    }
    return binarySearch(a, target, middle + 1, right);
}
```

Array search

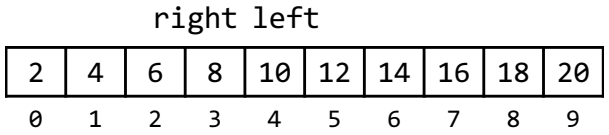
Now to describe the base case for each algorithm. Let's observe the stopping condition for both linear search and binary search:

The search ends unsuccessfully (returns -1) when there are no more array elements that remain to be examined; that is, when the search space is empty.

Linear search:



Binary search:



The base case, the smallest instance of the problem, is an empty search space, which is identified in the same way for both algorithms:

```
if (left > right) {
    return -1;
}
```

Array search

Linear search expressed recursively:

```
public static int linearSearch(int[] a, int target, int left, int right) {
    if (left > right) {
        return -1;
    }
    if (a[left] == target) {
        return left;
    }
    return linearSearch(a, target, left + 1, right);
}
```

Array search

Binary search expressed recursively:

```
public int binarySearch(int[] a, int target, int left, int right) {
    if (left > right) {
        return -1;
    }
    int middle = left + (right - left) / 2;
    if (a[middle] == target) {
        return middle;
    }
    if (a[middle] > target) {
        return binarySearch(a, target, left, middle - 1);
    }
    return binarySearch(a, target, middle + 1, right);
}
```