Recursion (Part 2)

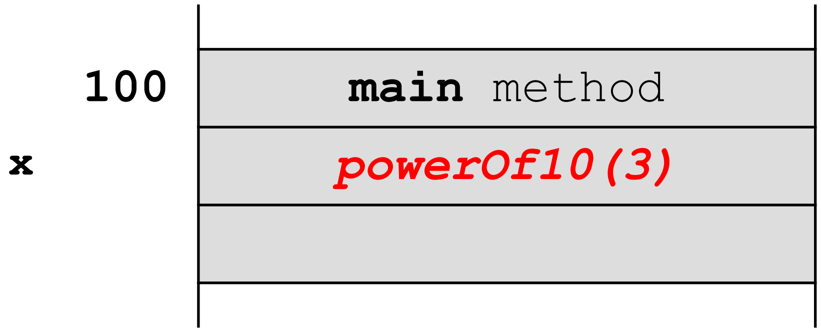
What Happens During Recursion?

- a simplified model of what happens during a recursive method invocation is the following:
  - whenever a method is invoked that method runs in a *new* block of memory
    - when a method recursively invokes itself, a new block of memory is allocated for the newly invoked method to run in

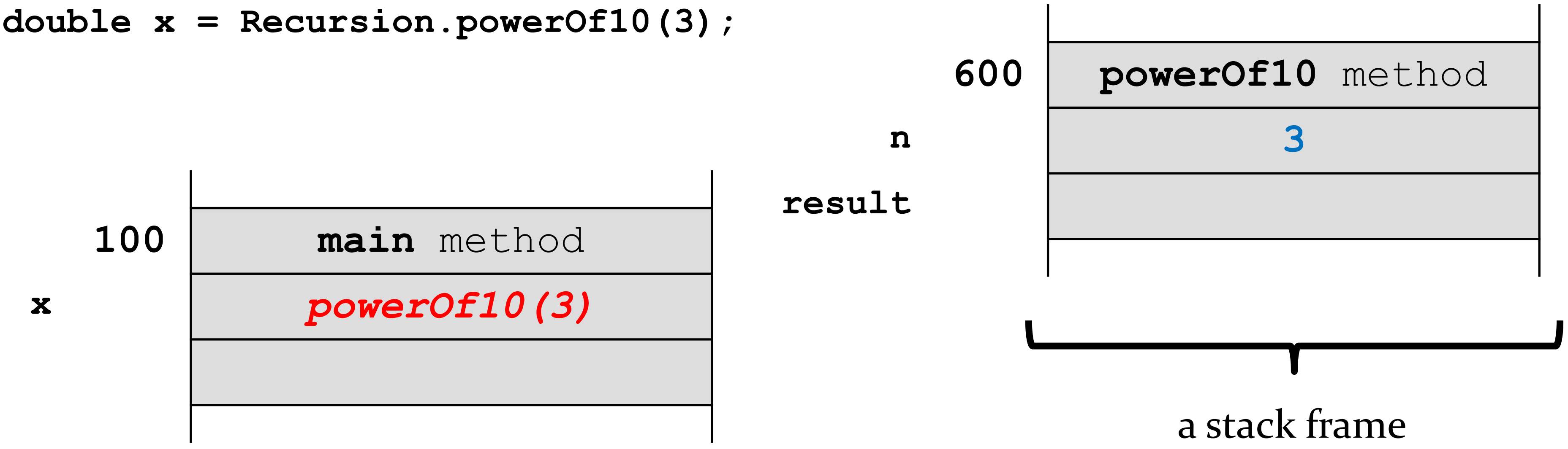
- consider a slightly modified version of the **`powerOf10`** method

```java
public static double powerOf10(int n) {
  double result;
  if (n < 0) {
    result = 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    result = 1.0;
  }
  else {
    result = 10 * powerOf10(n - 1);
  }
  return result;
}
```

```
double x = Recursion.powerOf10(3);
```

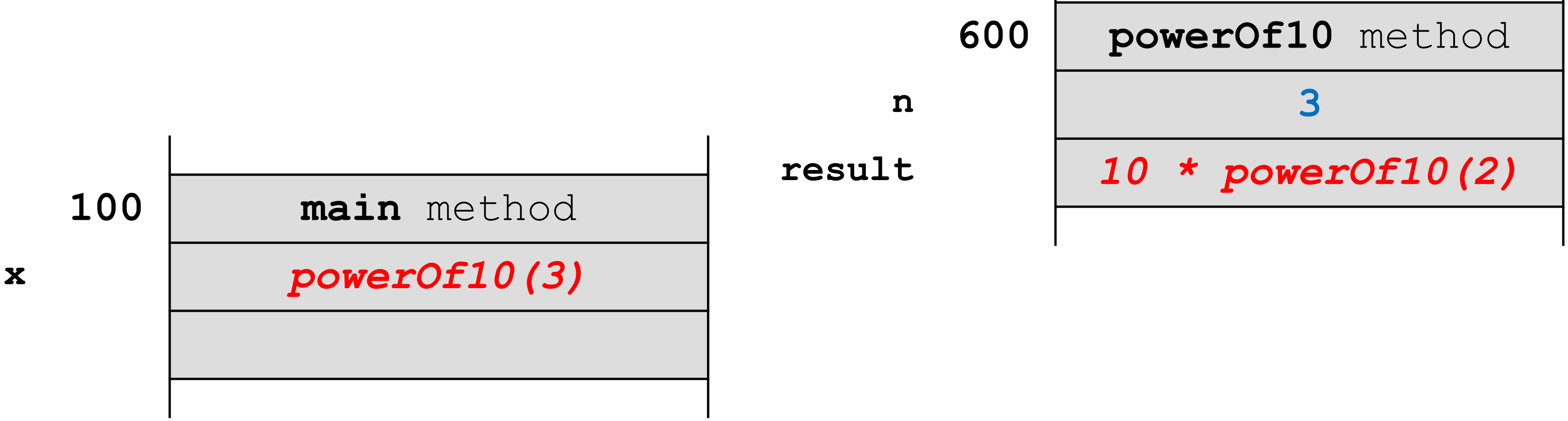


```
double x = Recursion.powerOf10(3);
```

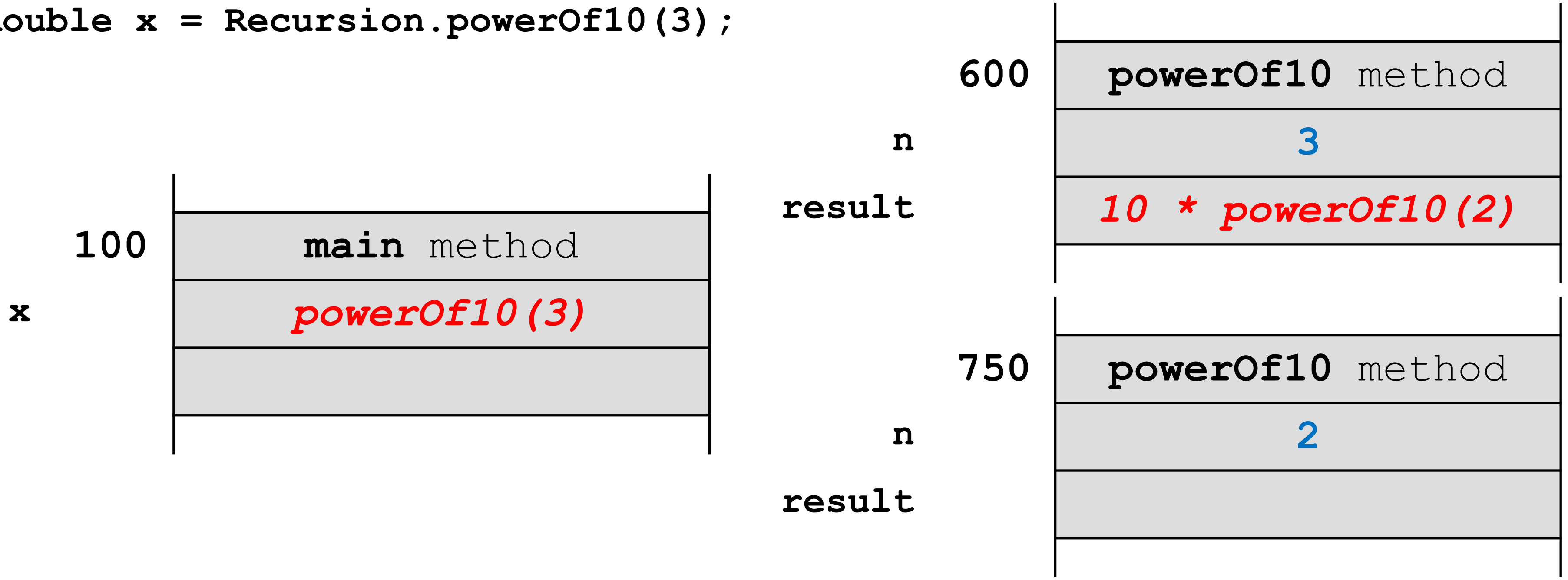


- methods occupy space in a region of memory called the *call stack*
- information regarding the state of the method is stored in a *stack frame*
- the stack frame includes information such as the method parameters, local variables of the method, where the return value of the method should be copied to, where control should flow to after the method completes, …
- stack memory can be allocated and deallocated very quickly, but this speed is obtained by restricting the total amount of stack memory
- if you try to solve a large problem using recursion you can exceed the available amount of stack memory which causes your program to crash

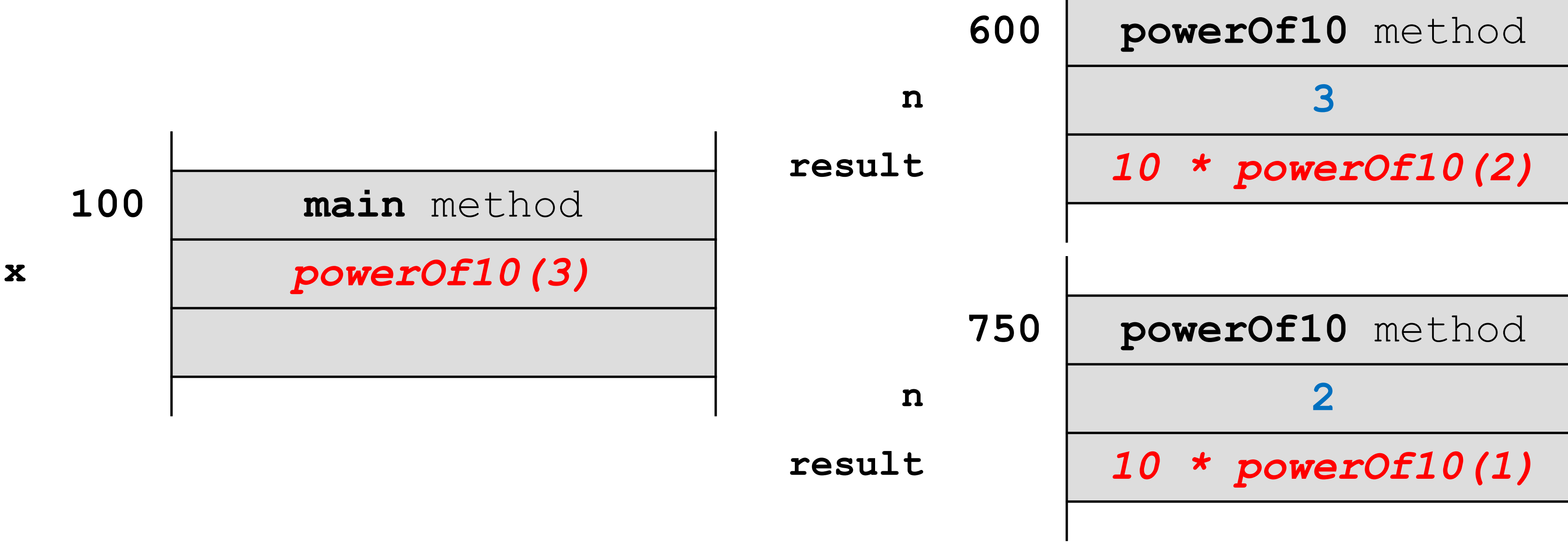
```
double x = Recursion.powerOf10(3);
```



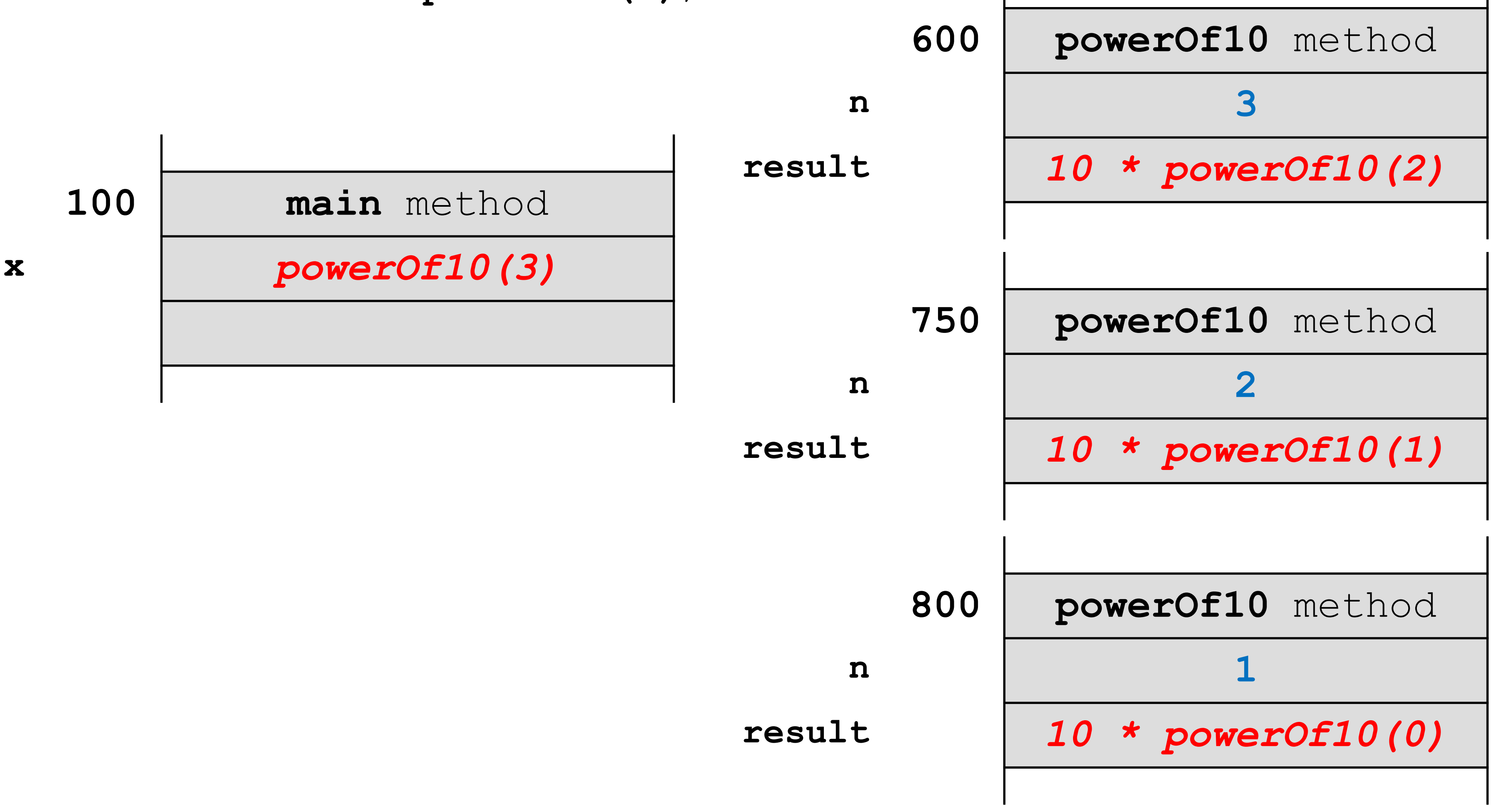
```
double x = Recursion.powerOf10(3);
```



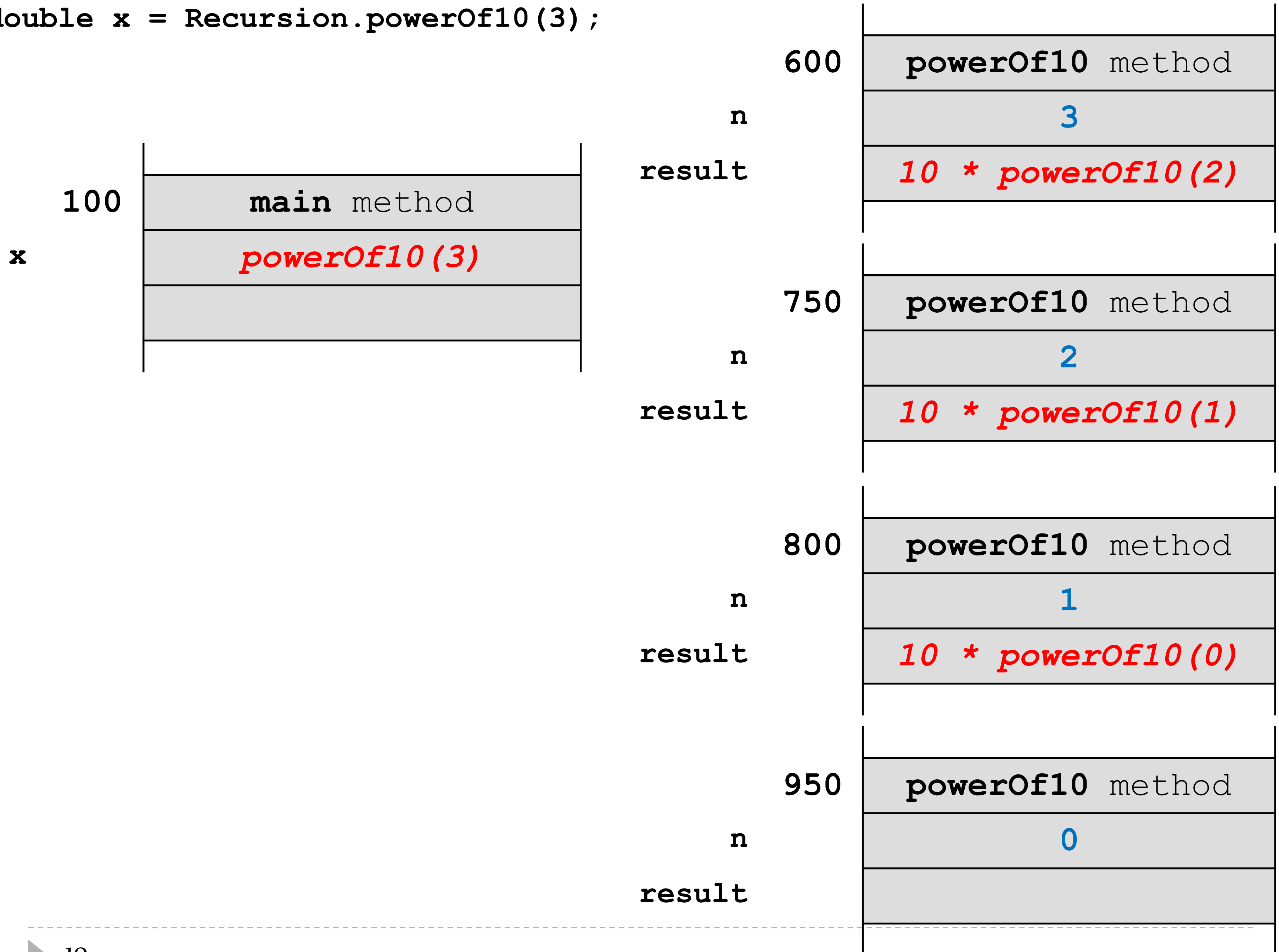
```
double x = Recursion.powerOf10(3);
```



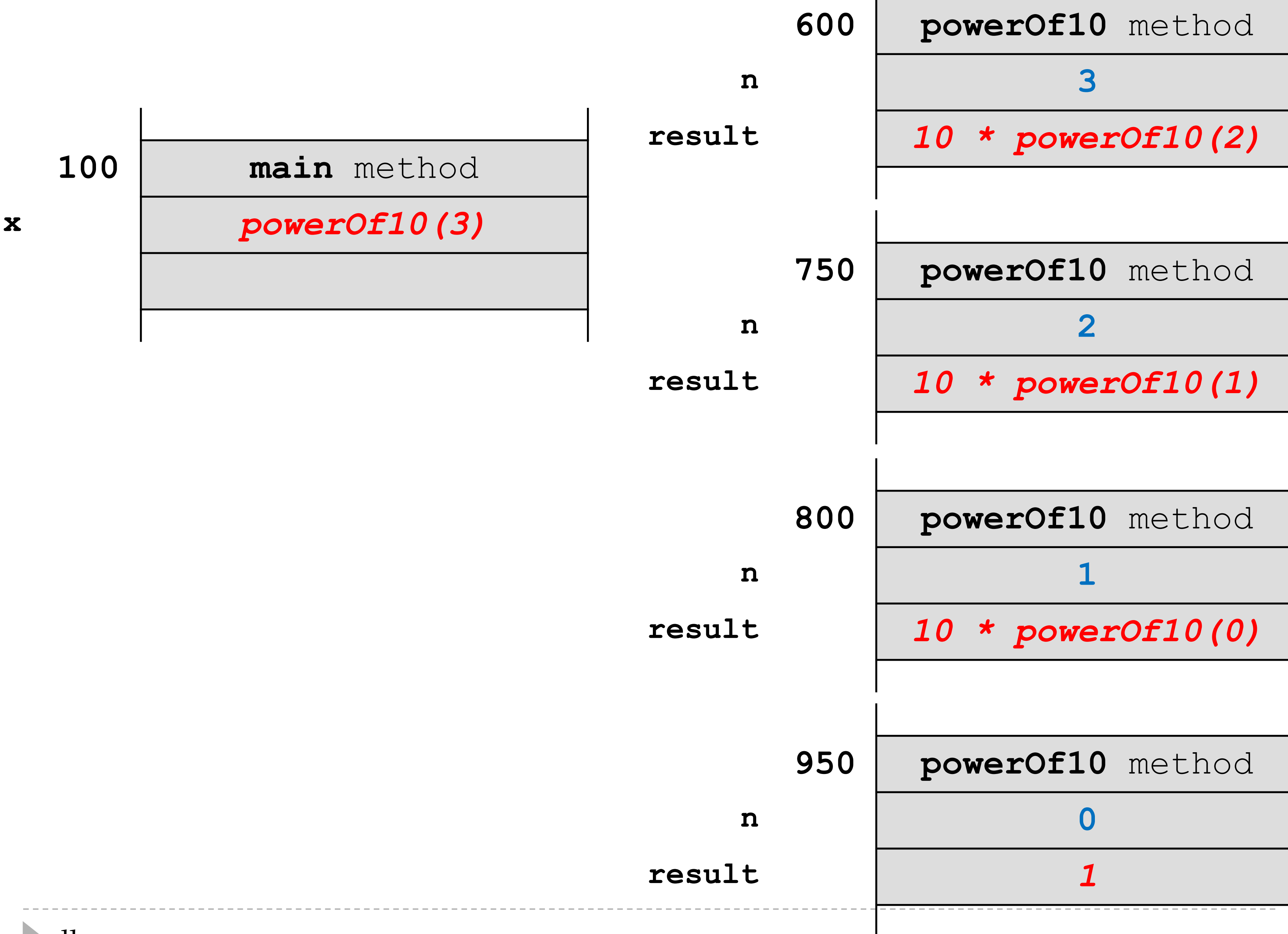
```
double x = Recursion.powerOf10(3);
```



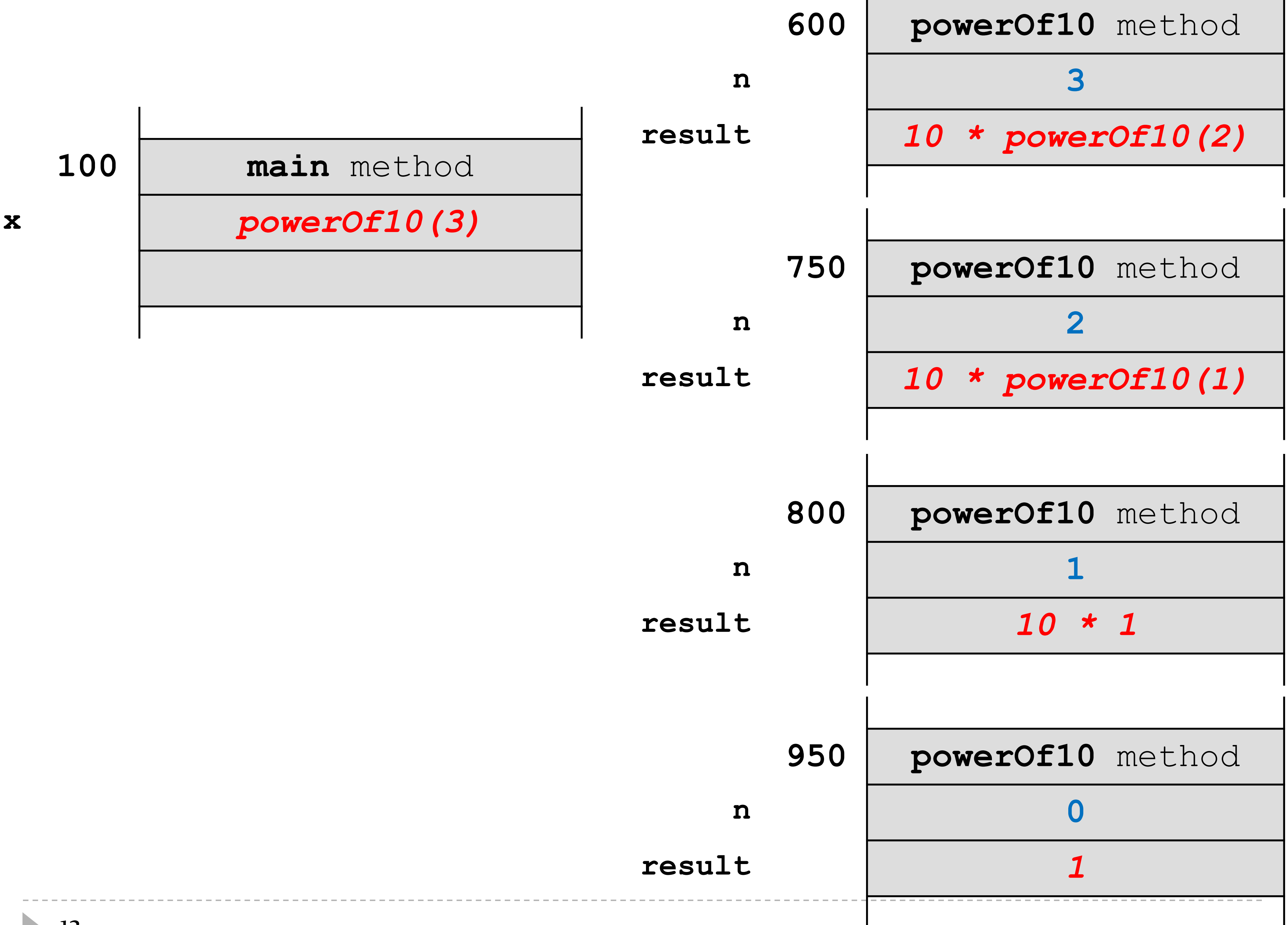
```
double x = Recursion.powerOf10(3);
```



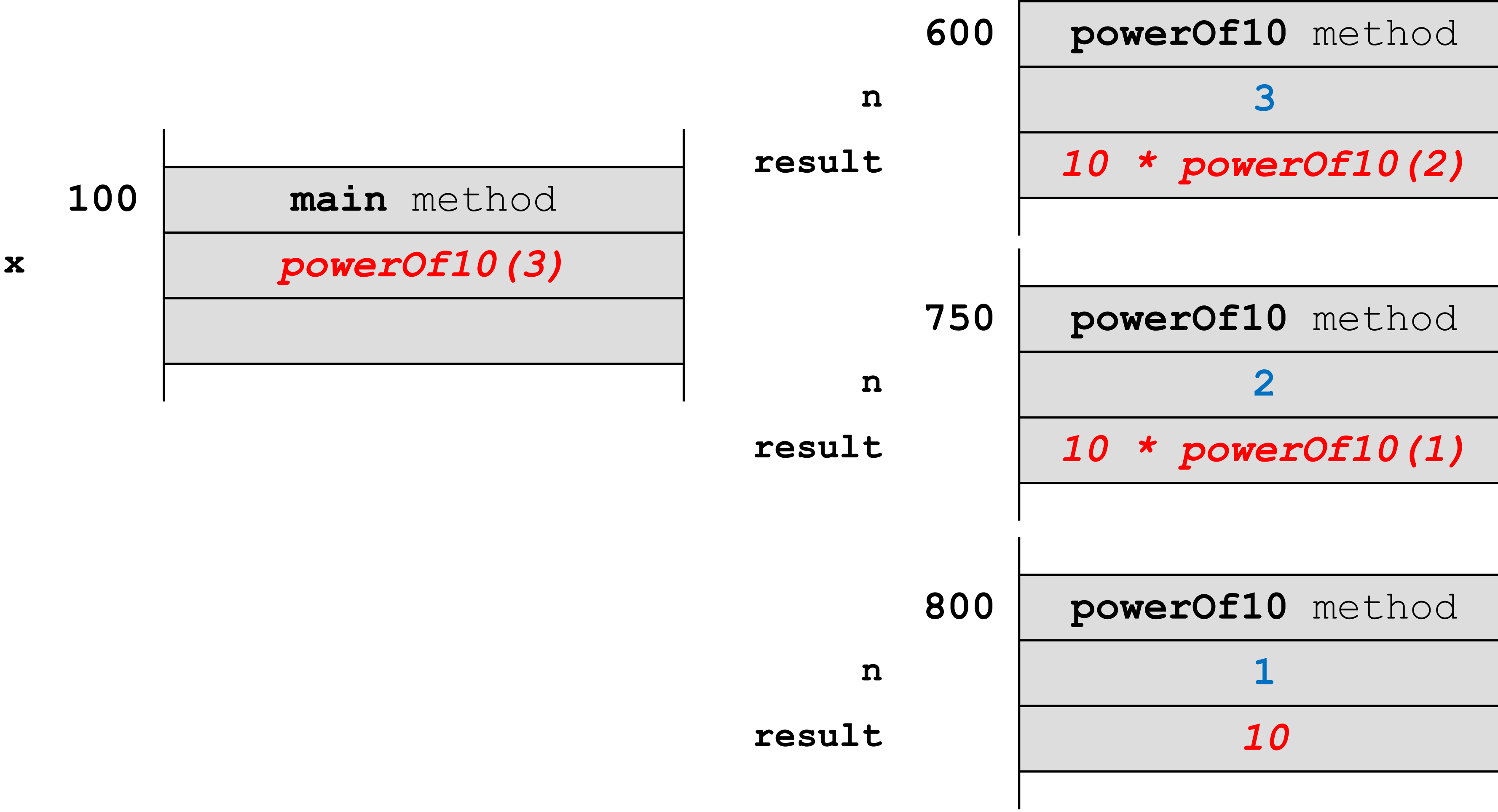
```
double x = Recursion.powerOf10(3);
```



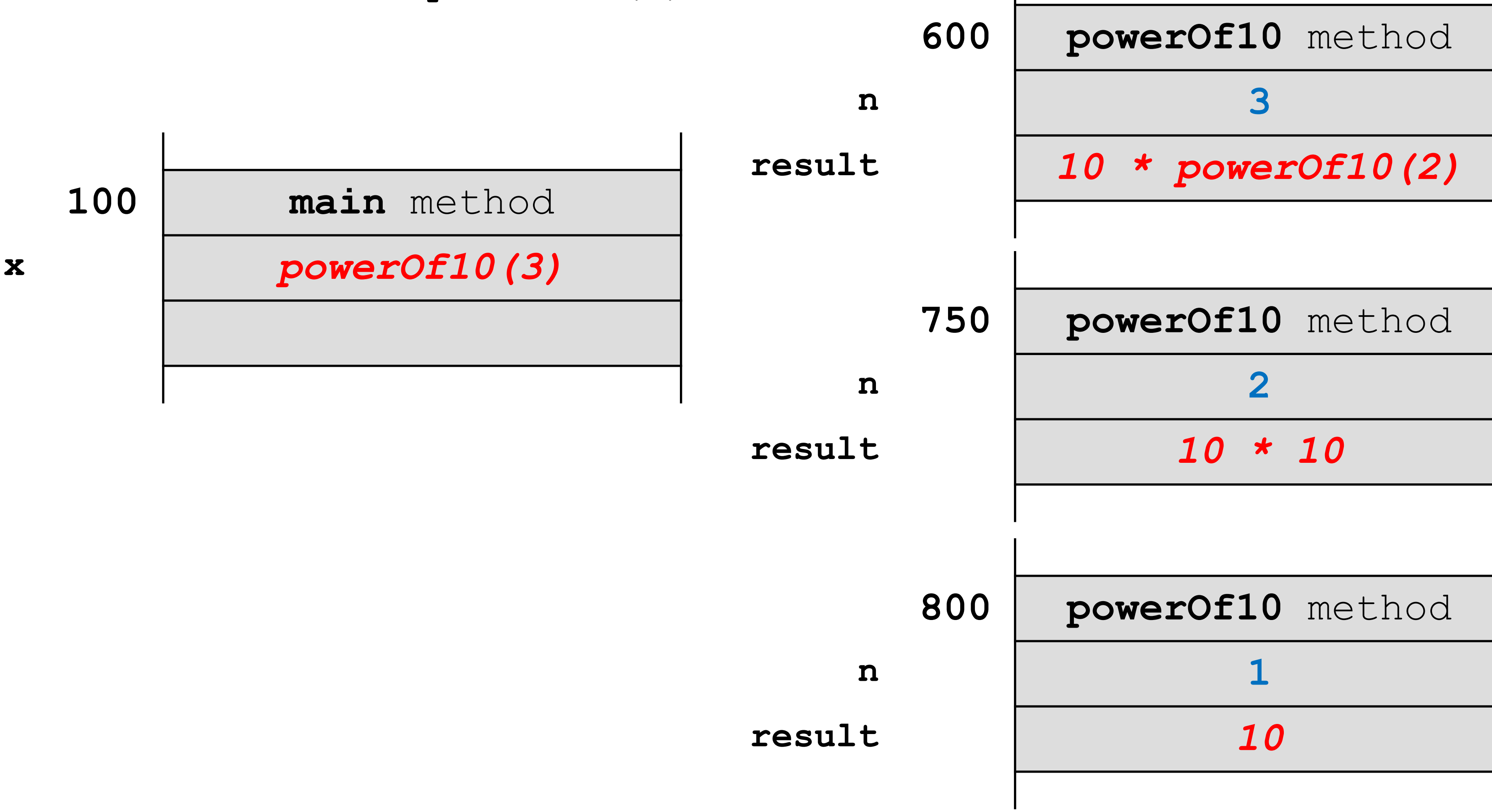
```
double x = Recursion.powerOf10(3);
```



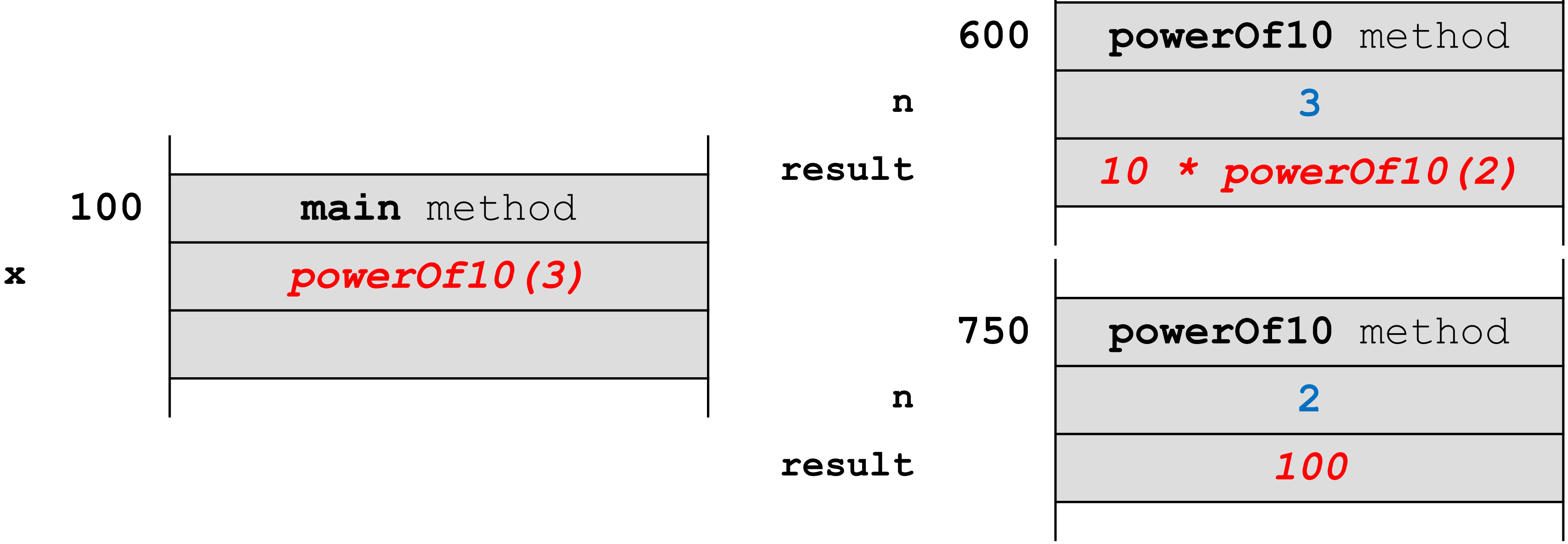
```
double x = Recursion.powerOf10(3);
```



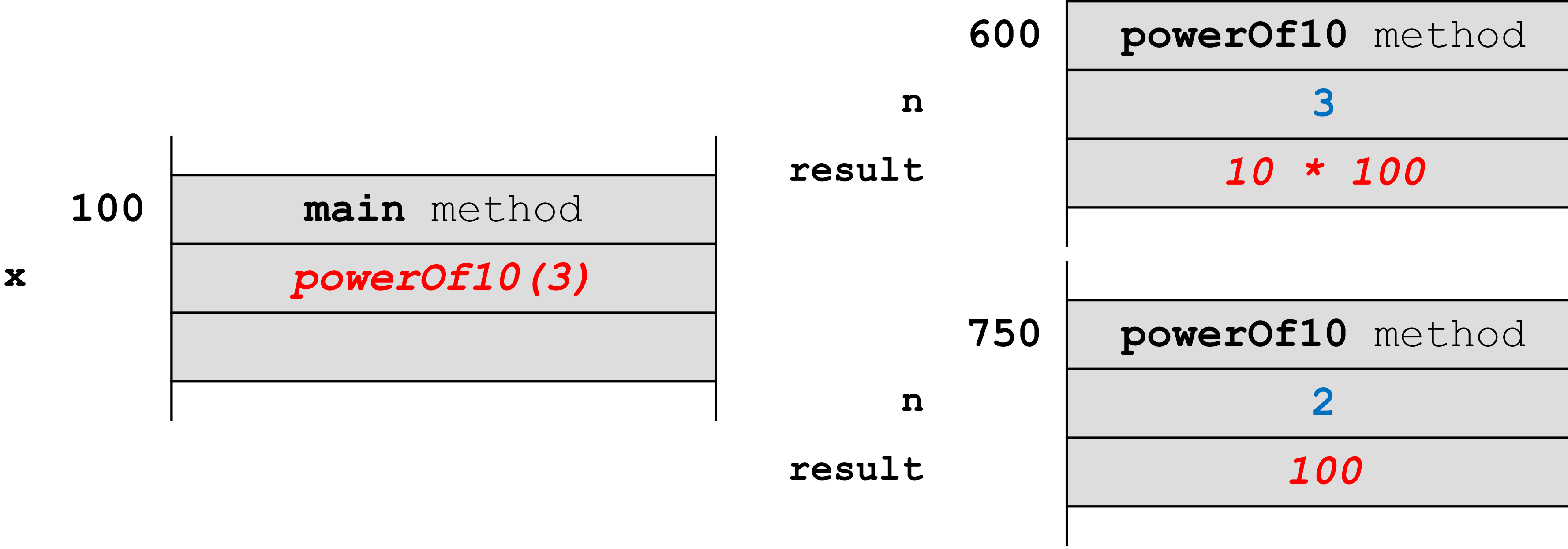
```
double x = Recursion.powerOf10(3);
```



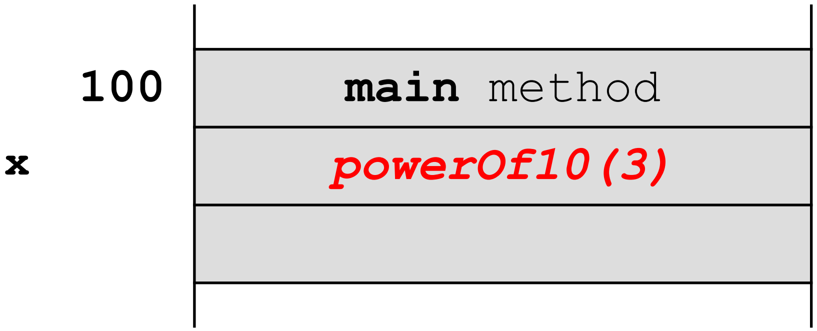
```
double x = Recursion.powerOf10(3);
```

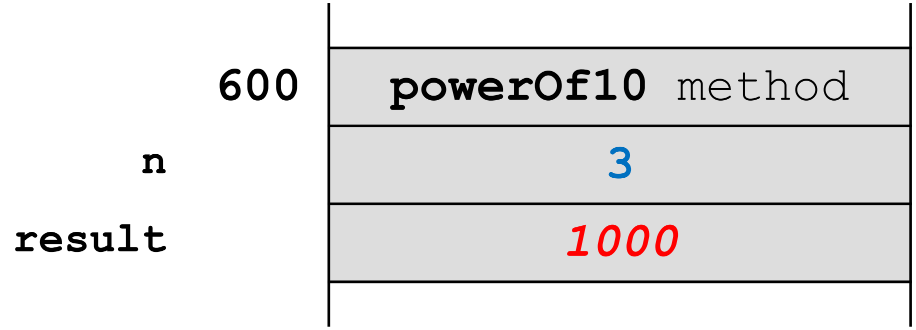


```
double x = Recursion.powerOf10(3);
```

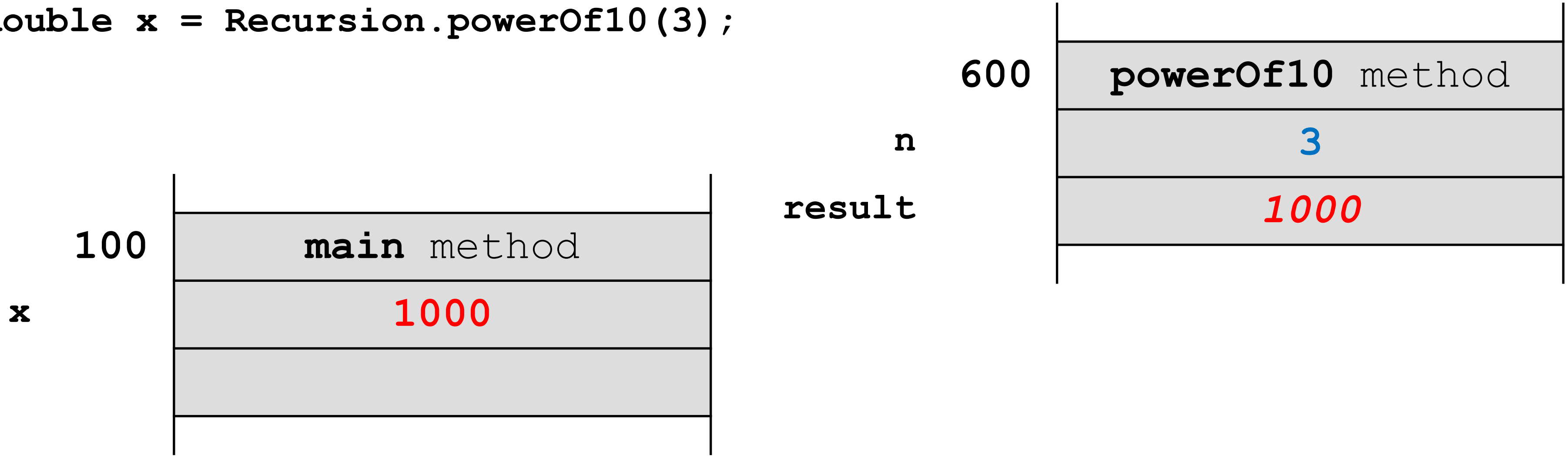


```
double x = Recursion.powerOf10(3);
```

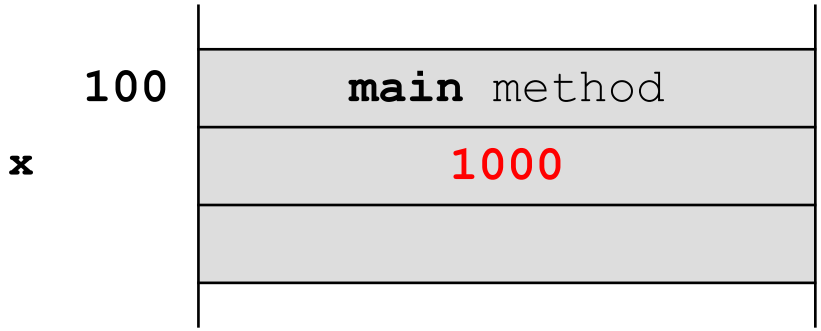




```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



Recursion and Collections

- consider the problem of searching for an element in a list
- searching a list for a particular element can be performed by recursively examining the first element of the list
  - if the first element is the element we are searching for then we can return true
  - otherwise, we recursively search the sub-list starting at the next element

The **List** method **subList**

- **List** has a very useful method named **subList**:

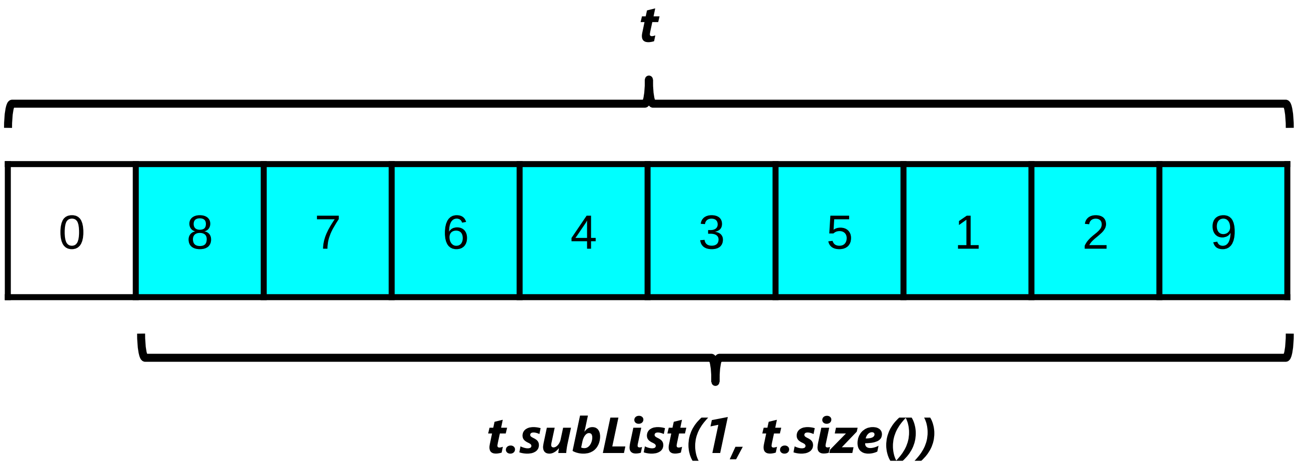
List<E> subList(int fromIndex, int toIndex)

Returns a view of the portion of this list between the specified **fromIndex**, inclusive, and **toIndex**, exclusive. (If **fromIndex** and **toIndex** are equal, the returned list is empty.) The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa. The returned list supports all of the optional list operations supported by this list.

http://docs.oracle.com/javase/7/docs/api/java/util/List.html#subList%28int,%20int%29

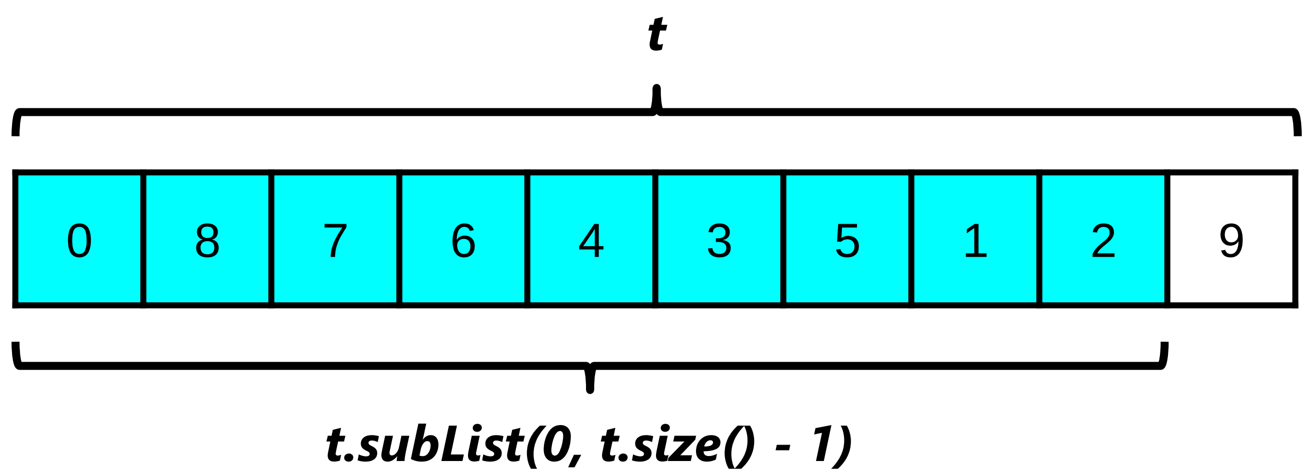
`subList` examples

- the sub-list excluding the first element of the original list



`subList` examples

- the sub-list excluding the last element of the original list



Recursively Search a List

```
  contains("X", ["Z", "Q", "B", "X", "J"])

→ "X".equals("Z") == false
→ contains("X", ["Q", "B", "X", "J"])  recursive call

→ "X".equals("Q") == false
→ contains("X", ["B", "X", "J"])       recursive call

→ "X".equals("B") == false
→ contains("X", ["X", "J"])            recursive call

→ "X".equals("X") == true              done!
```

Recursively Search a List

- base case(s)?
  - recall that a base case occurs when the solution to the problem is known

```java
public class Week10 {

  public static <T> boolean contains(T element, List<T> t) {
    boolean result;




  }
}
```

Recursively Search a List

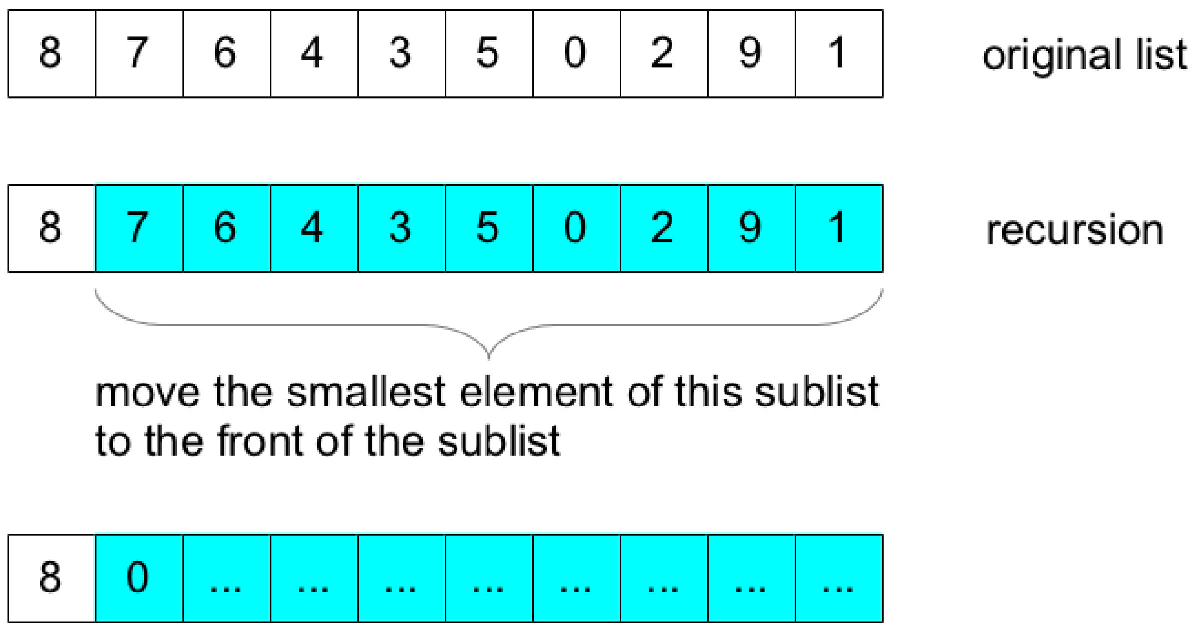
- recursive call?
  - to help deduce the recursive call assume that the method does exactly what its API says it does
    - e.g., **`contains(element, t)`** returns true if **`element`** is in the list **`t`** and false otherwise
  - use the assumption to write the recursive call or calls

```java
public class Week10 {

  public static <T> boolean contains(T element, List<T> t) {
    boolean result;




  }
}
```

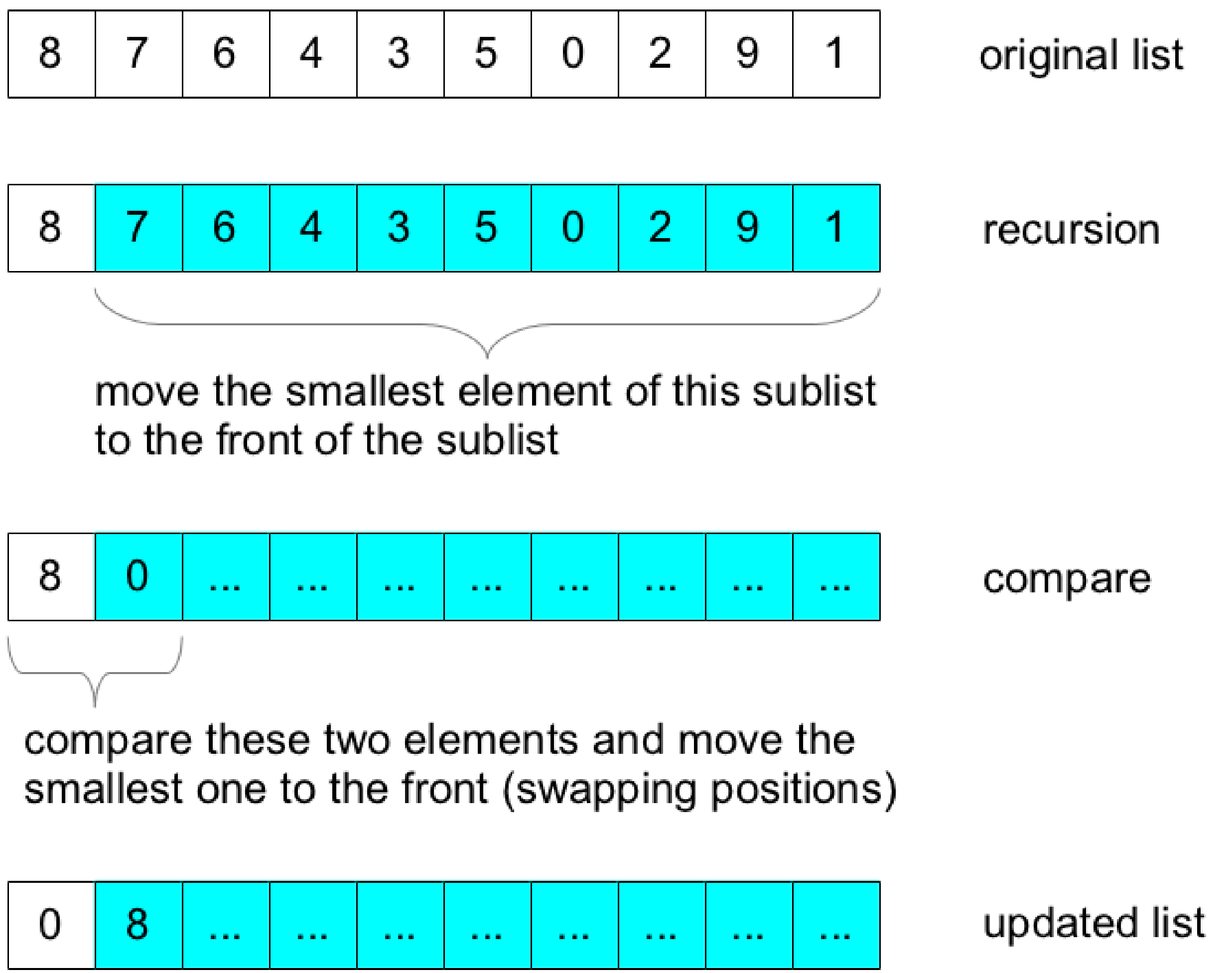
Recursion and Collections

- consider the problem of moving the smallest element in a list of integers to the front of the list

Recursively Move Smallest to Front



Recursively Move Smallest to Front



Recursively Move Smallest to Front

- base case?
  - recall that a base case occurs when the solution to the problem is known

Recursively Move Smallest to Front

```
public class Week10 {

  public static void minToFront(List<Integer> t) {






  }
}
```

Recursively Move Smallest to Front

- recursive call?
  - to help deduce the recursive call assume that the method does exactly what its API says it does
    - e.g., **`moveToFront(t)`** moves the smallest element in **`t`** to the front of **`t`**
    - use the assumption to write the recursive call or calls

Recursively Move Smallest to Front

```
public class Week10 {

  public static void minToFront(List<Integer> t) {

  }
}
```

Recursively Move Smallest to Front

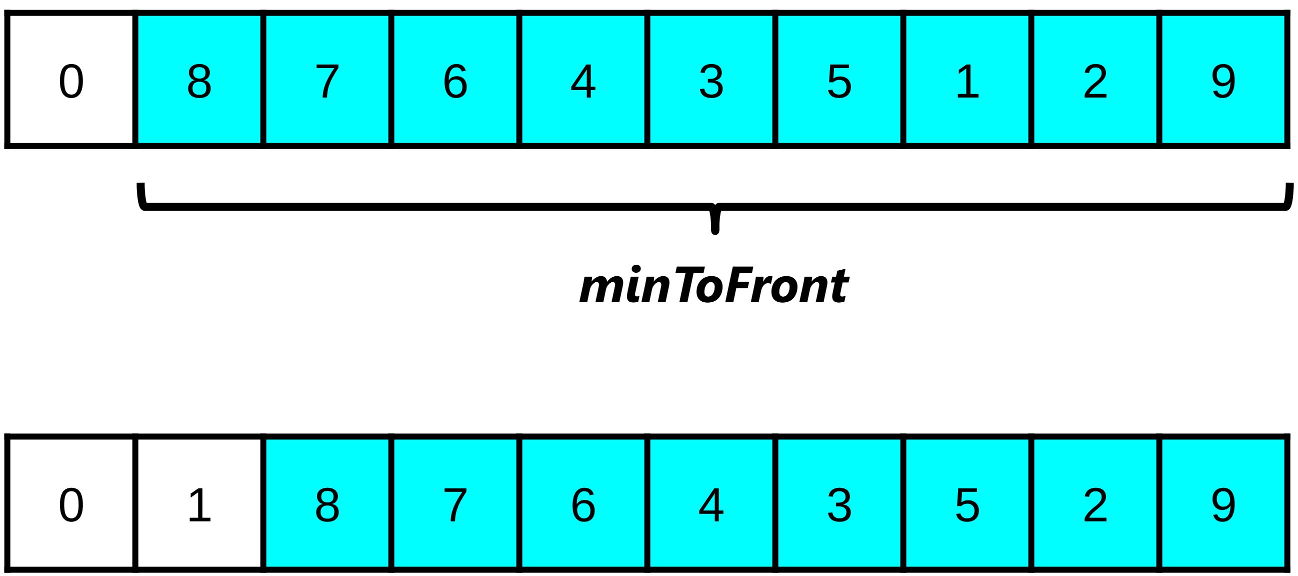
- compare and update?

Recursively Move Smallest to Front

```
public class Week10 {

  public static void minToFront(List<Integer> t) {




  }
}
```

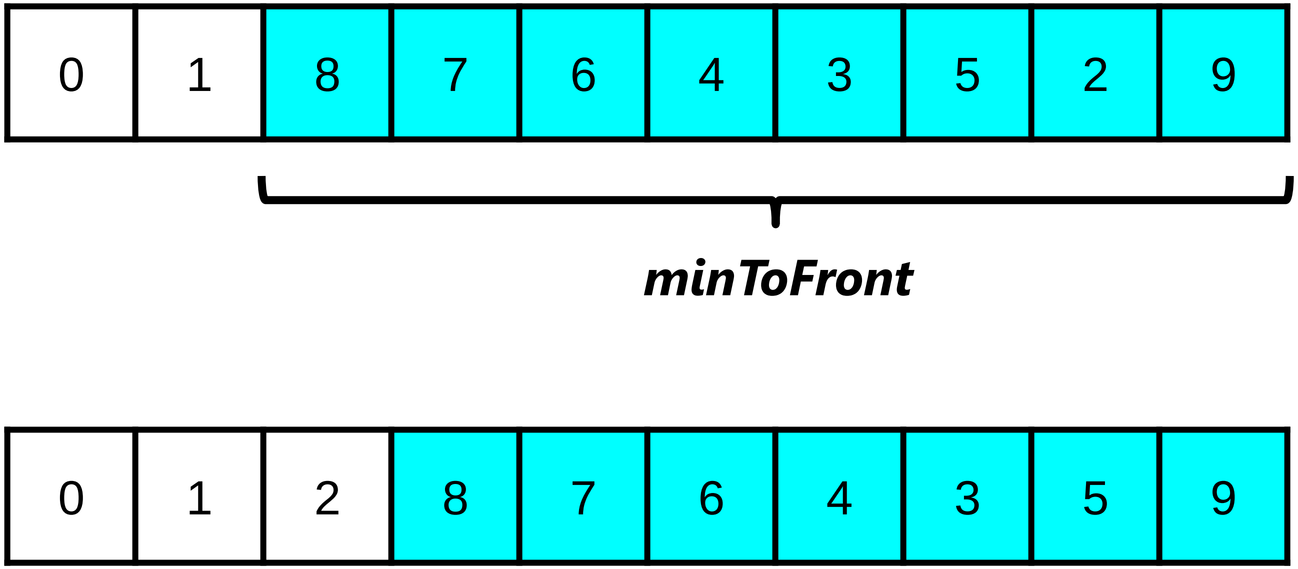
Sorting the List

- observe what happens if you repeat the process with the sublist made up of the second through last elements:



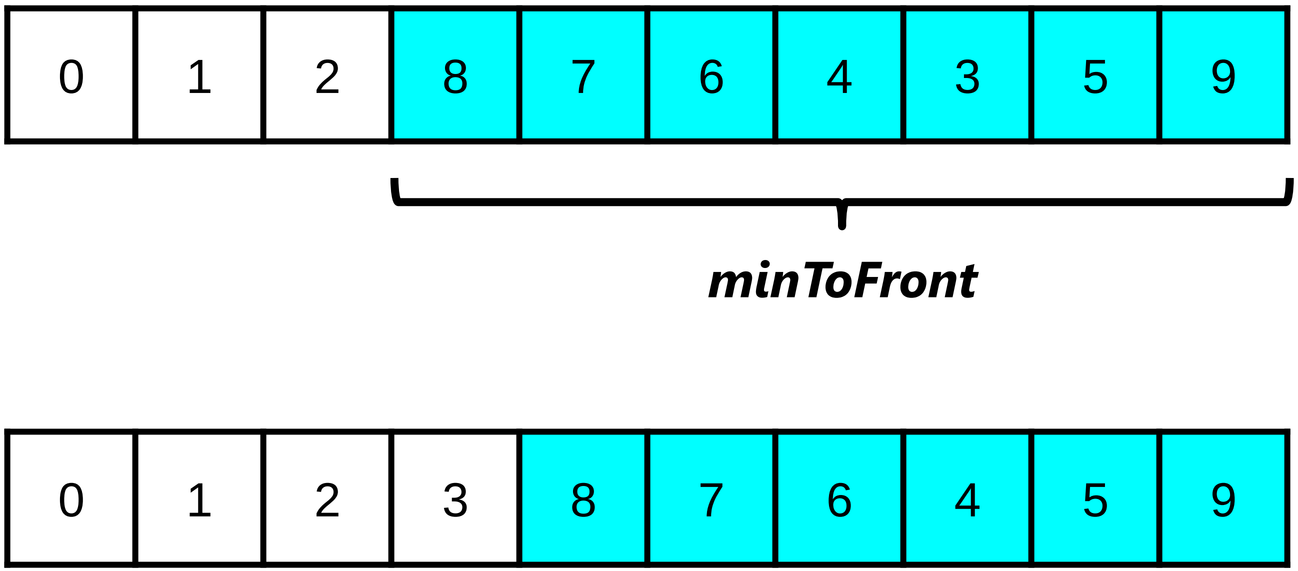
Sorting the List

- observe what happens if you repeat the process with the sublist made up of the third through last elements:



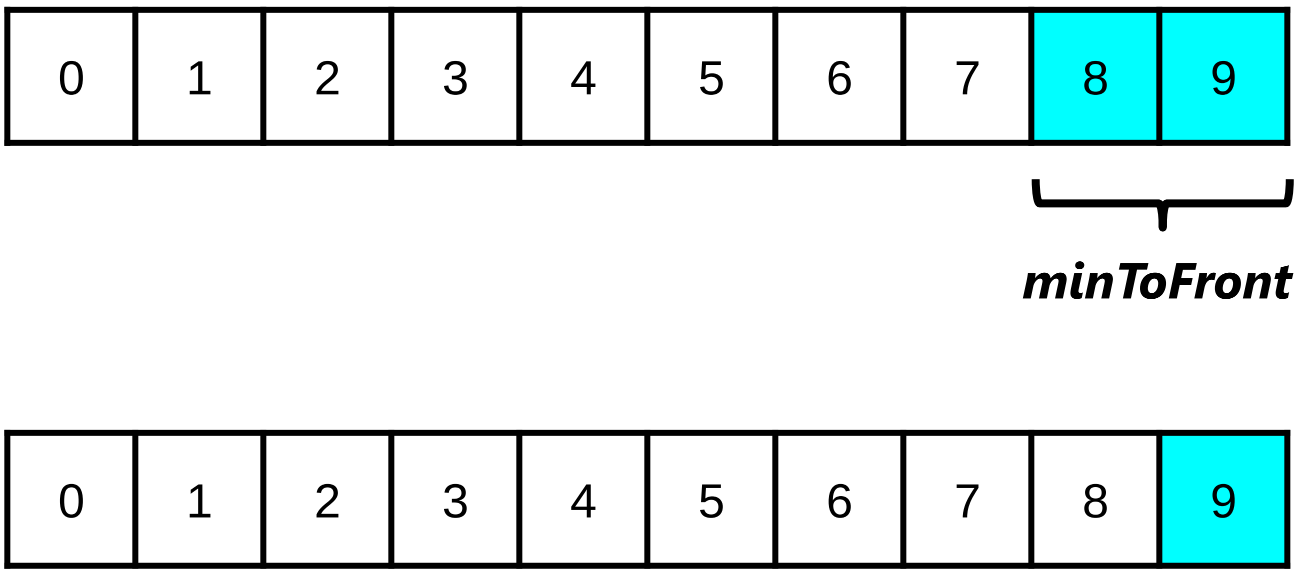
Sorting the List

- observe what happens if you repeat the process with the sublist made up of the fourth through last elements:



Sorting the List

- if you keep calling **`minToFront`** until you reach a sublist of size two, you will sort the original list:



- this is the *selection sort* algorithm

Selection Sort

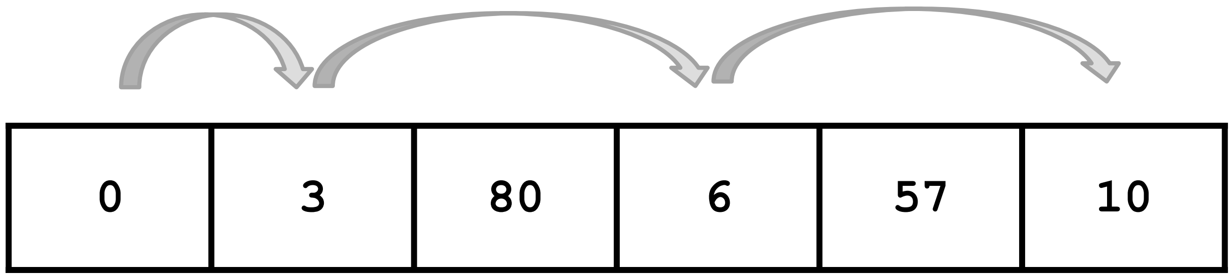
```
public class Sort {

  // minToFront not shown

  public static void selectionSort(List<Integer> t) {
    if (t.size() > 1) {
      Sort.minToFront(t);
      Sort.selectionSort(t.subList(1, t.size()));
    }
  }


}
```

Jump It

0	3	80	6	57	10

- board of n squares, n >= 2
- start at the first square on left
- on each move you can move 1 *or* 2 squares to the right
- each square you land on has a cost (the value in the square)
  - costs are always positive
- goal is to reach the rightmost square with the lowest cost

Jump It

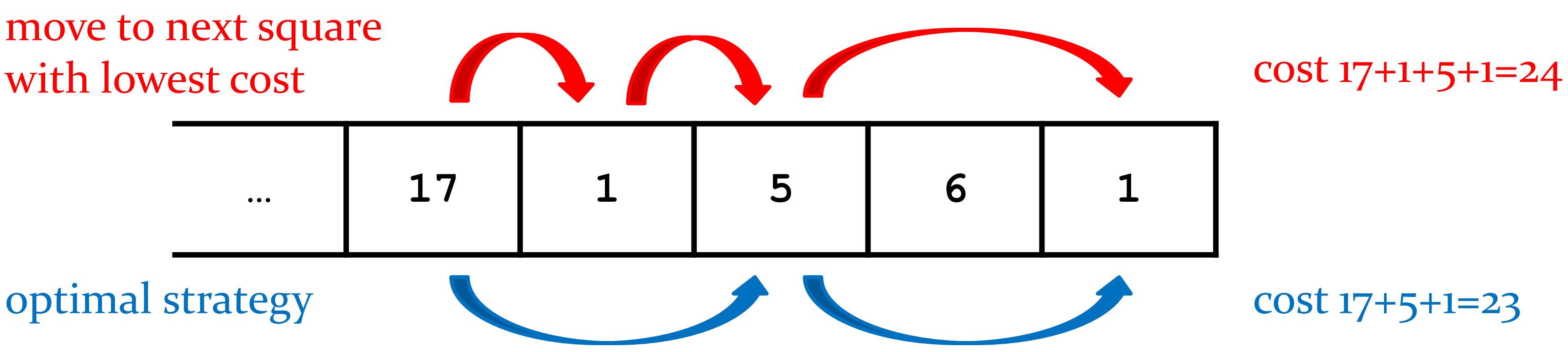


- solution for example:
  - move 1 square
  - move 2 squares
  - move 2 squares
    - total cost = 0 + 3 + 6 + 10 = 19

- can the problem be solved by always moving to the next square with the lowest cost?

Jump It

- no, it might be better to move to a square with higher cost because you would have ended up on that square anyway



Jump It

- sketch a small example of the problem
  - it will help you find the base cases
  - it might help you find the recursive cases

Jump It

- base case(s):
  - **`board.size() == 2`**
    - no choice of move (must move 1 square)
    - **`cost = board.get(0) + board.get(1);`**
  - **`board.size() == 3`**
    - move 2 squares (avoiding the cost of 1 square)
    - **`cost = board.get(0) + board.get(2);`**

Jump It

```java
public static int cost(List<Integer> board) {
  if (board.size() == 2) {
    return board.get(0) + board.get(1);
  }
  if (board.size() == 3) {
    return board.get(0) + board.get(2);
  }




}
```

Jump It

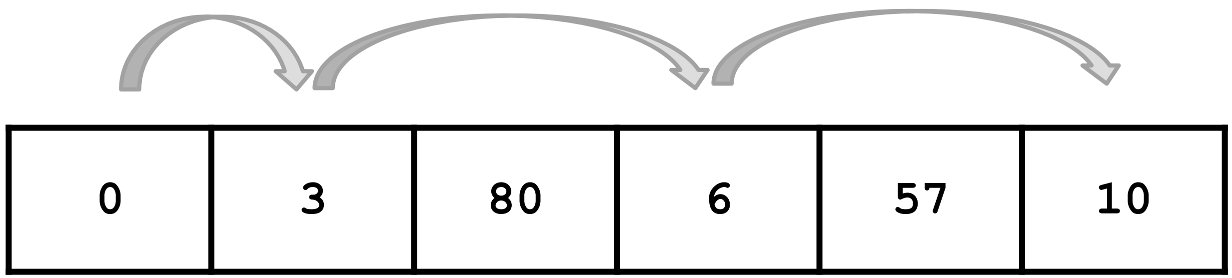
- recursive case(s):
  - compute the cost of moving 1 square
  - compute the cost of moving 2 squares
- return the smaller of the two costs

Jump It

```
public static int cost(List<Integer> board) {
  if (board.size() == 2) {
    return board.get(0) + board.get(1);
  }
  if (board.size() == 3) {
    return board.get(0) + board.get(2);
  }
  List<Integer> afterOneStep = board.subList(1, board.size());
  List<Integer> afterTwoStep = board.subList(2, board.size());
  int c = board.get(0);
  return c + Math.min(cost(afterOneStep), cost(afterTwoStep));
}
```

Jump It

- can you modify the `cost` method so that it also produces a list of moves?
  - e.g., for the following board



  the method produces the list `[1, 2, 2]`
- consider using the following modified signature

```
public static int cost(List<Integer> board, List<Integer> moves)
```

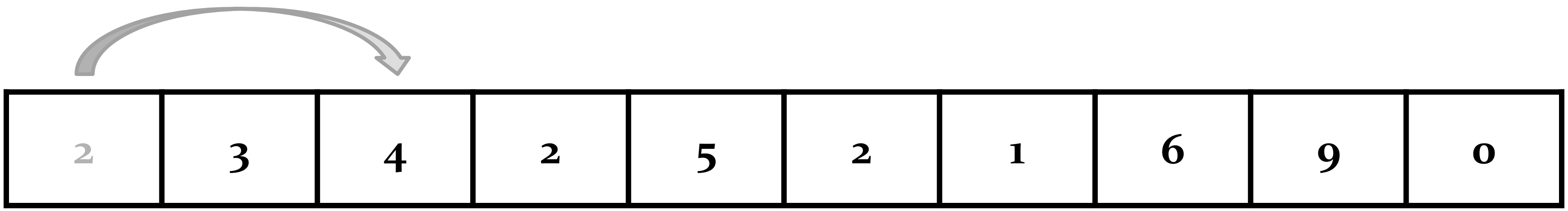
- the Jump It problem has a couple of nice properties:
  - the rules of the game make it impossible to move to the same square twice
  - the rules of the games make it impossible to try to move off of the board
- consider the following problem

- given a list of non-negative integer values:

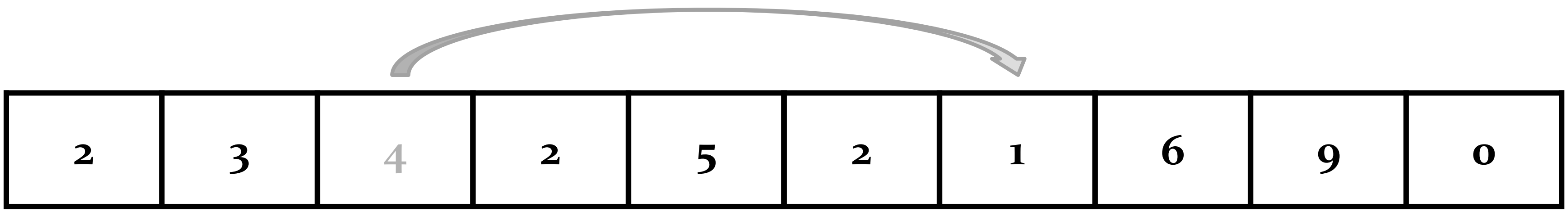
2	3	4	2	5	2	1	6	9	0

  - starting from the first element try to reach the last element (whose value is always zero)
  - you may move left or right by the number of elements equal to the value of the element that you are currently on
  - you may not move outside the bounds of the list

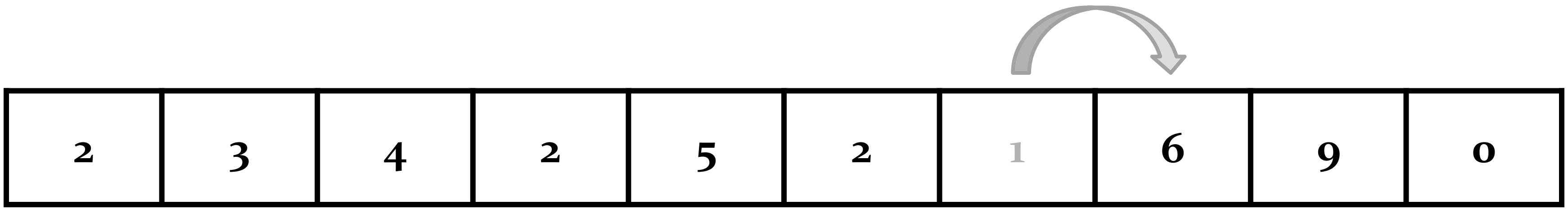
Solution 1



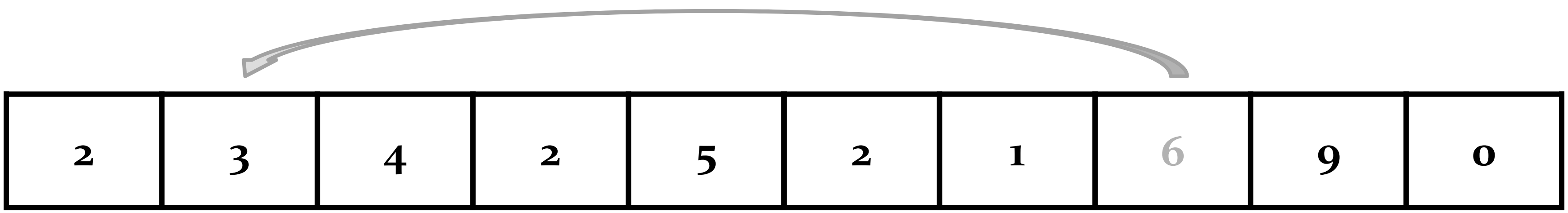
Solution 1



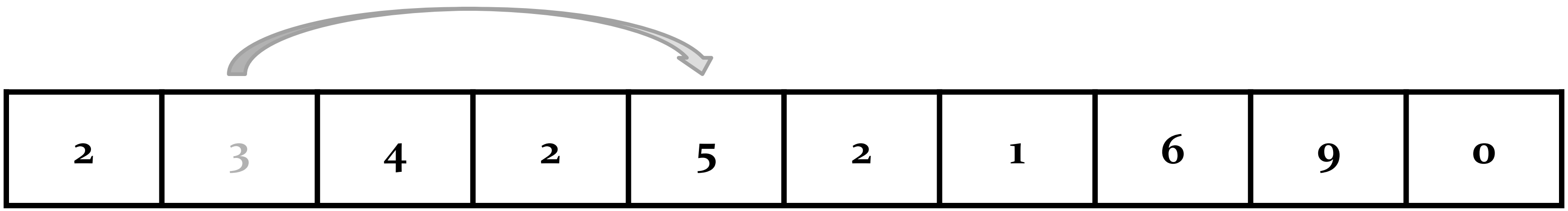
Solution 1



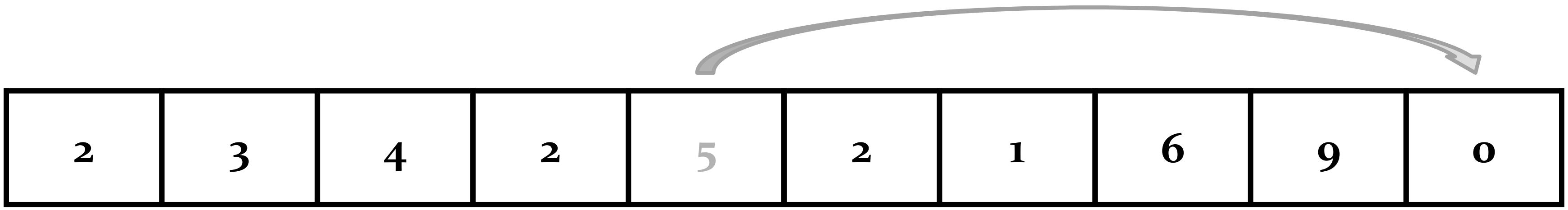
Solution 1



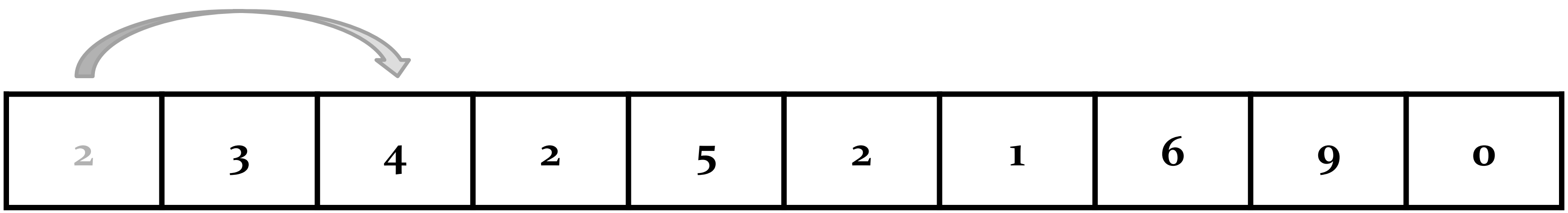
Solution 1



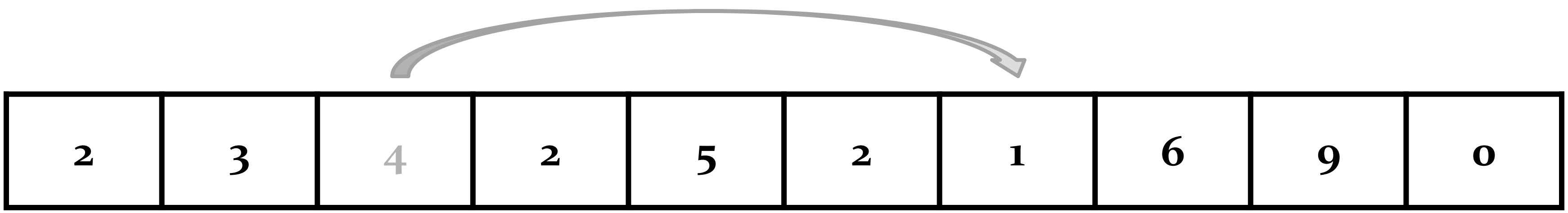
Solution 1



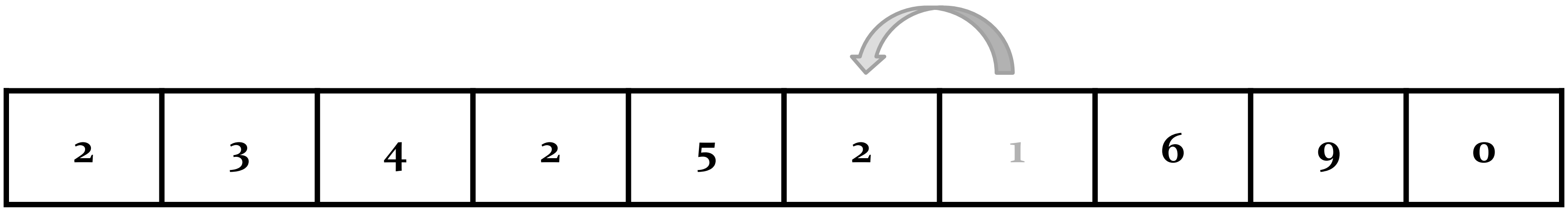
Solution 2



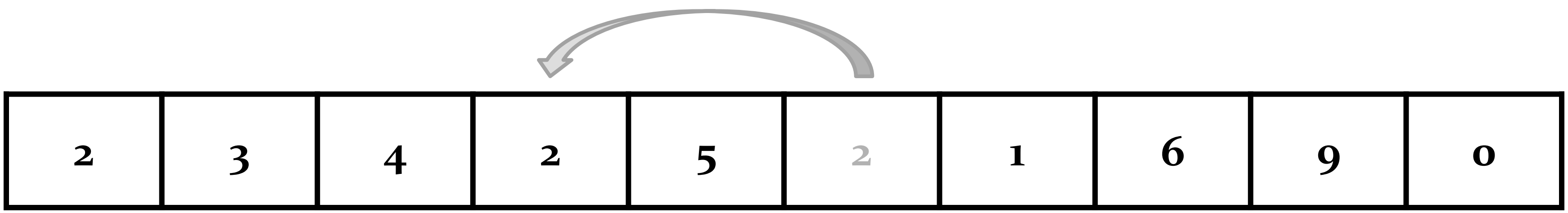
Solution 2



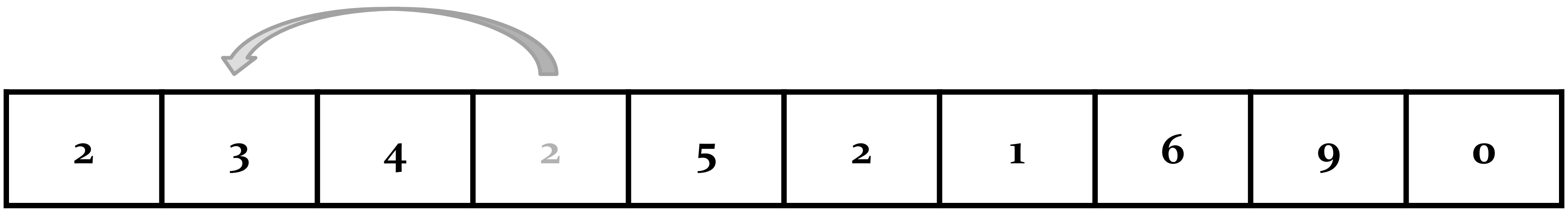
Solution 2



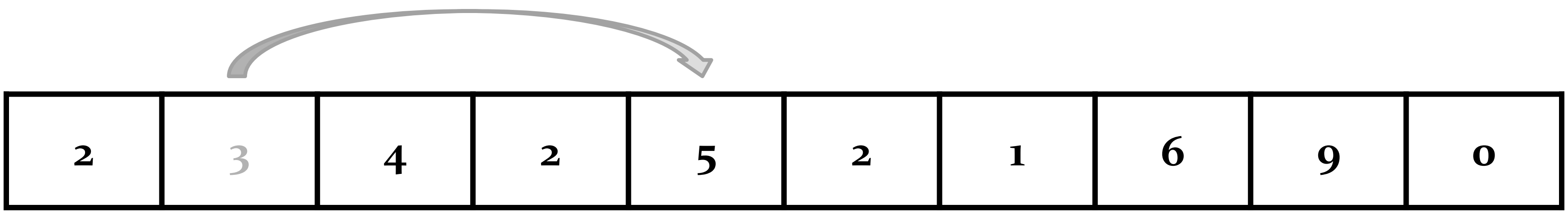
Solution 2



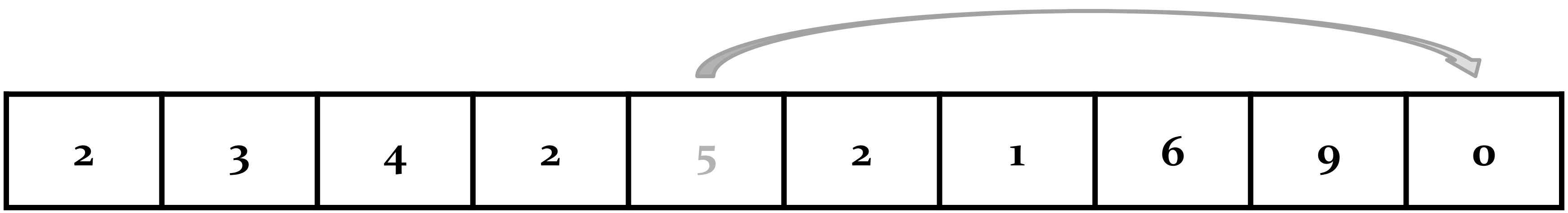
Solution 2



Solution 2

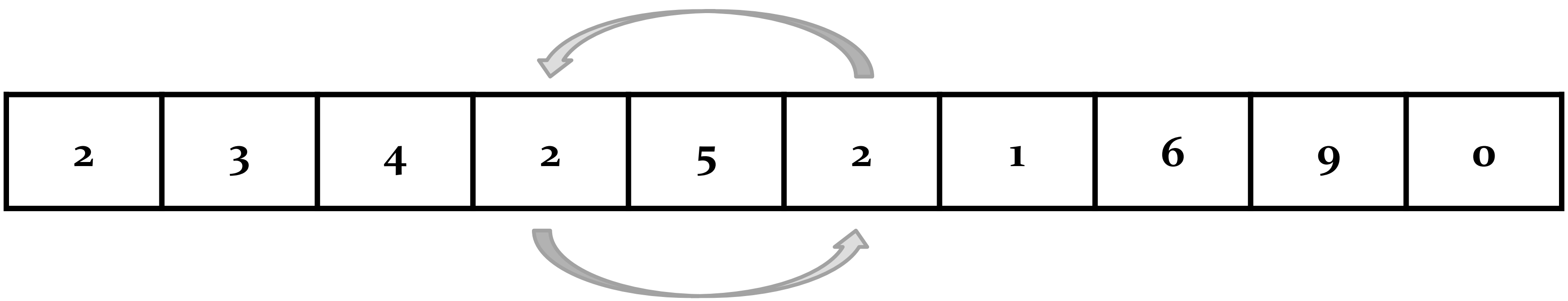


Solution 2



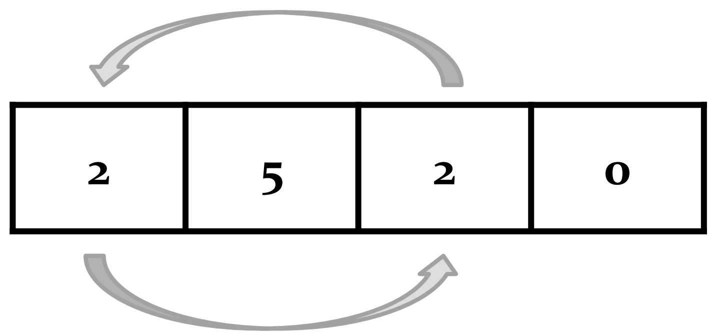
Cycles

- it is possible to find cycles where a move takes you back to a square that you have already visited



Cycles

- using a cycle, it is easy to create a board where no solution exists



Cycles

- on the board below, no matter what you do, you eventually end up on the 1 which leads to a cycle

2	1	2	2	2	2	2	6	3	0

No Solution

- even without using a cycle, it is easy to create a board where no solution exists

1	100	2	0

- unlike Jump It, the board does not get smaller in an obvious way after each move
  - but it does in fact get smaller (otherwise, a recursive solution would never terminate)
    - how does the board get smaller?
    - how do we indicate this?

Recursion

- recursive cases:
  - can we move left without falling off of the board?
    - if so, can the board be solved by moving to the left?
  - can we move right without falling off of the board?
    - if so, can the board be solved by moving to the right?

```java
/**
 * Is a board is solvable when the current move is at location
 * index of the board? The method does not modify the board.
 *
 * @param index
 *          the current location on the board
 * @param board
 *          the board
 * @return true if the board is solvable, false otherwise
 */
public static boolean isSolvable(int index, List<Integer> board) {
}
```

```java
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;








}
```

```java
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;
    if ((index - value) >= 0) {

    }








}
```

```
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;
    if ((index - value) >= 0) {
      winLeft = isSolvable(index - value, copy);
    }









}
```

```java
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;
    if ((index - value) >= 0) {
      winLeft = isSolvable(index - value, copy);
    }

    copy = new ArrayList<Integer>(board);
    copy.set(index, -1);

}
```

```java
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;
    if ((index - value) >= 0) {
      winLeft = isSolvable(index - value, copy);
    }

    copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winRight = false;
    if ((index + value) < board.size()) {

    }

}
```

```java
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;
    if ((index - value) >= 0) {
      winLeft = isSolvable(index - value, copy);
    }

    copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winRight = false;
    if ((index + value) < board.size()) {
      winRight = isSolvable(index + value, copy);
    }

}
```

```java
public static boolean isSolvable(int index, List<Integer> board) {
    // base cases here
    int value = board.get(index);
    List<Integer> copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winLeft = false;
    if ((index - value) >= 0) {
      winLeft = isSolvable(index - value, copy);
    }

    copy = new ArrayList<Integer>(board);
    copy.set(index, -1);
    boolean winRight = false;
    if ((index + value) < board.size()) {
      winRight = isSolvable(index + value, copy);
    }
    return winLeft || winRight;
}
```

works, but does a lot of unnecessary computation; can you improve on this solution?

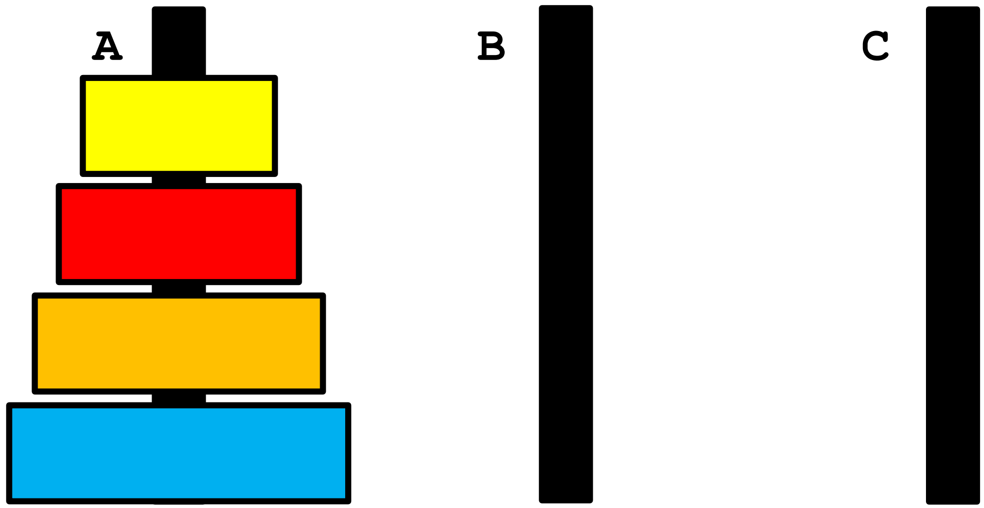
Base Cases

- base cases:
  - we've reached the last square
    - board is solvable
  - we've reached a square whose value is -1
    - board is not solvable

```
public static boolean isSolvable(int index, List<Integer> board) {
    if (board.get(index) < 0) {
      return false;
    }
    if (index == board.size() - 1) {
      return true;
    }
    // recursive cases go here...









}
```

Towers of Hanoi

- a problem easily solved using recursion



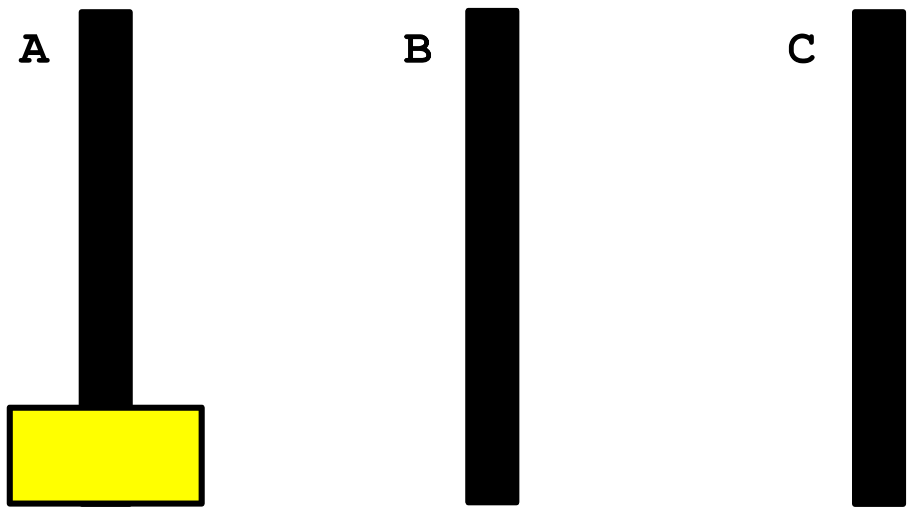
  - move the stack of *n* disks from A to C
    - can move one disk at a time from the top of one stack onto another stack
    - cannot move a larger disk onto a smaller disk

Towers of Hanoi

- legend says that the world will end when a 64 disk version of the puzzle is solved
- several appearances in pop culture
  - Doctor Who
  - Rise of the Planet of the Apes
  - Survior: South Pacific

Towers of Hanoi

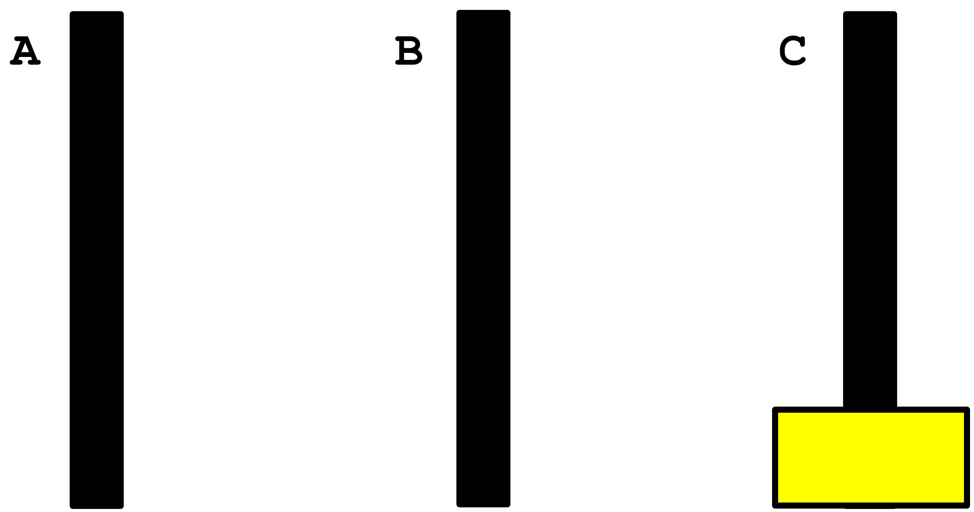
- n = 1



- move disk from A to C

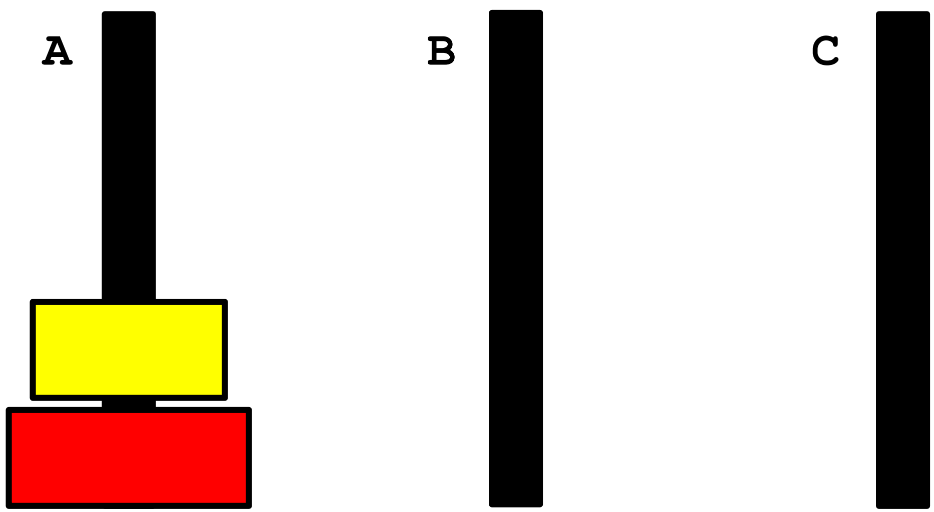
Towers of Hanoi

- n = 1



Towers of Hanoi

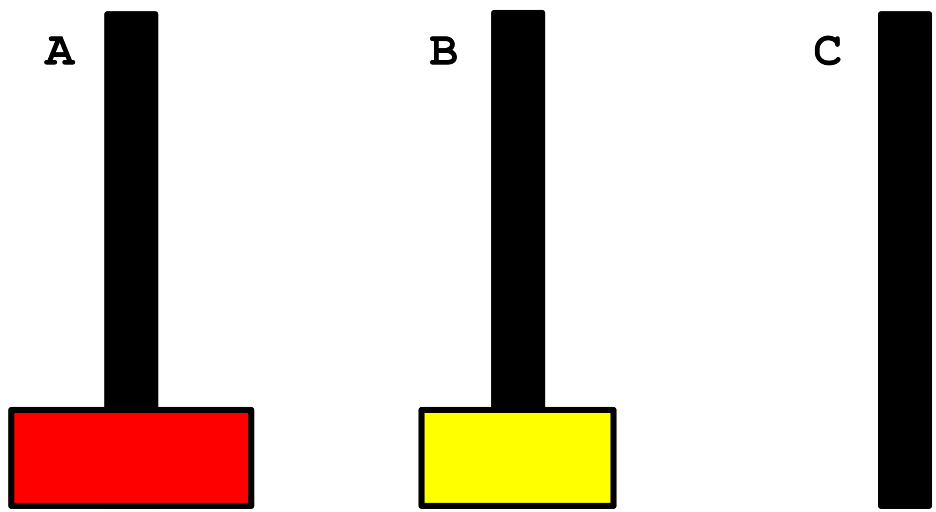
- n = 2



- move disk from A to B

Towers of Hanoi

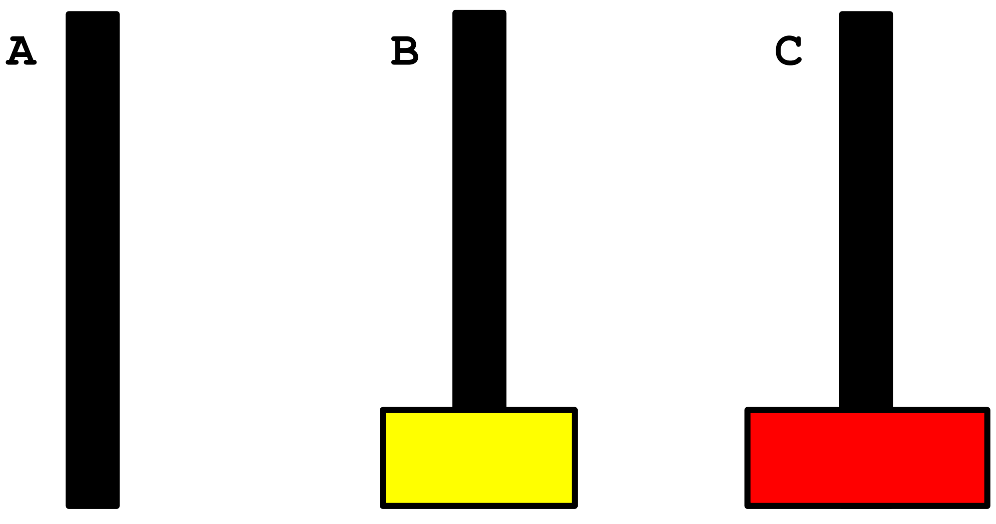
- n = 2



- move disk from A to C

Towers of Hanoi

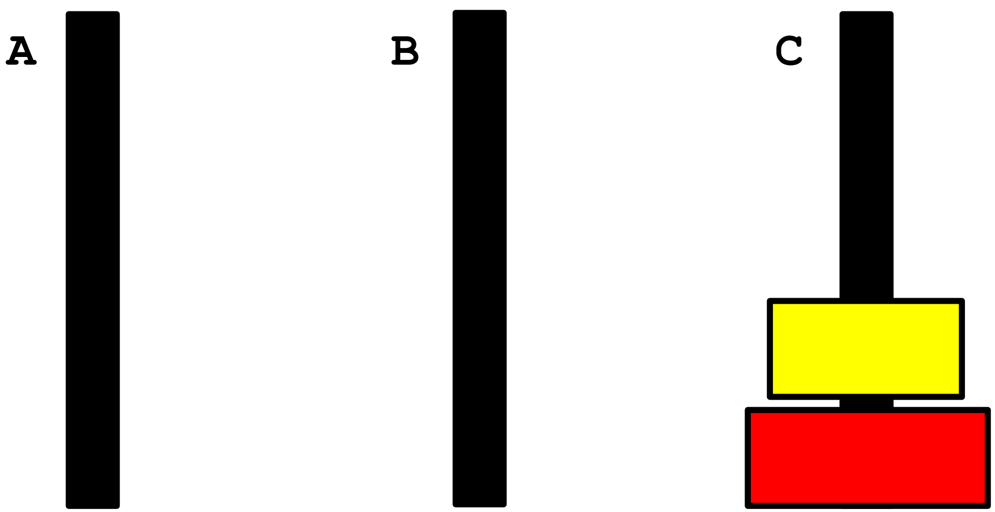
- n = 2



- move disk from B to C

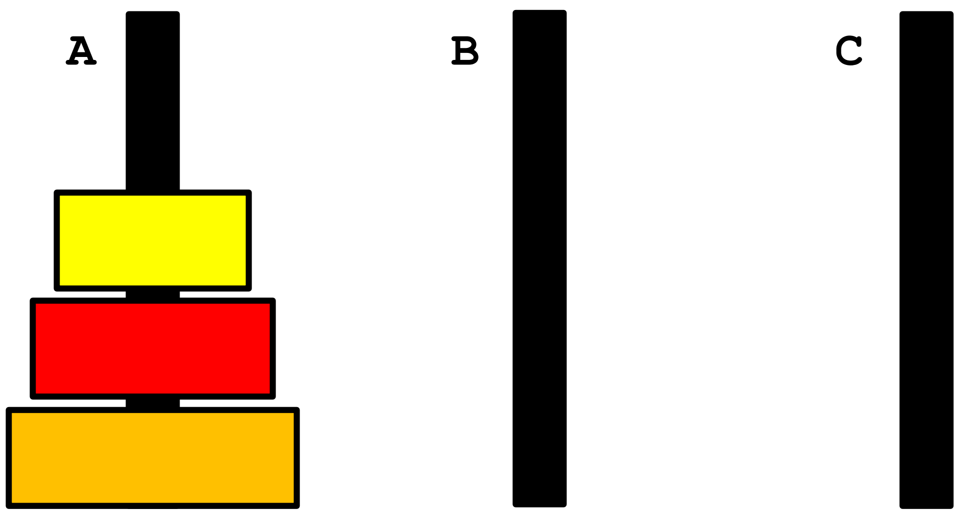
Towers of Hanoi

- n = 2



Towers of Hanoi

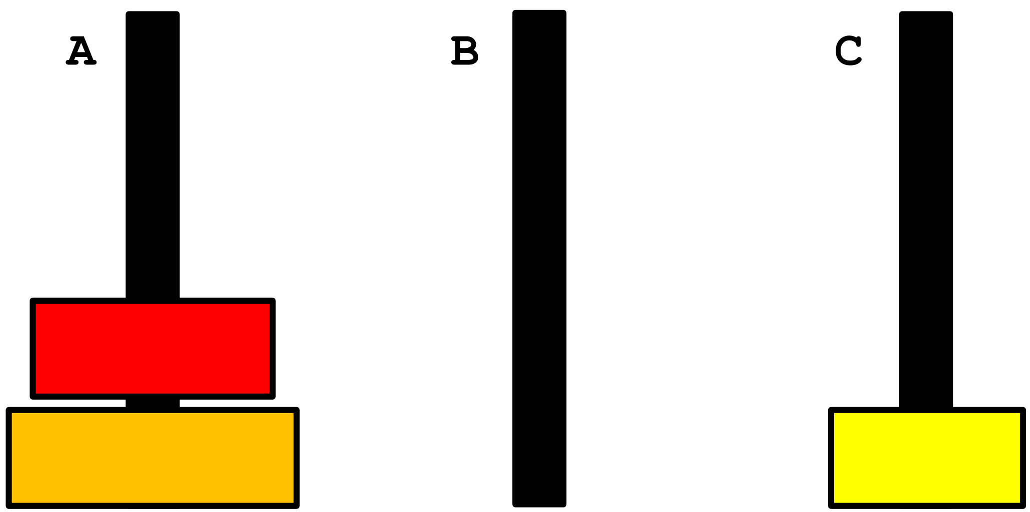
- n = 3



- move disk from A to C

Towers of Hanoi

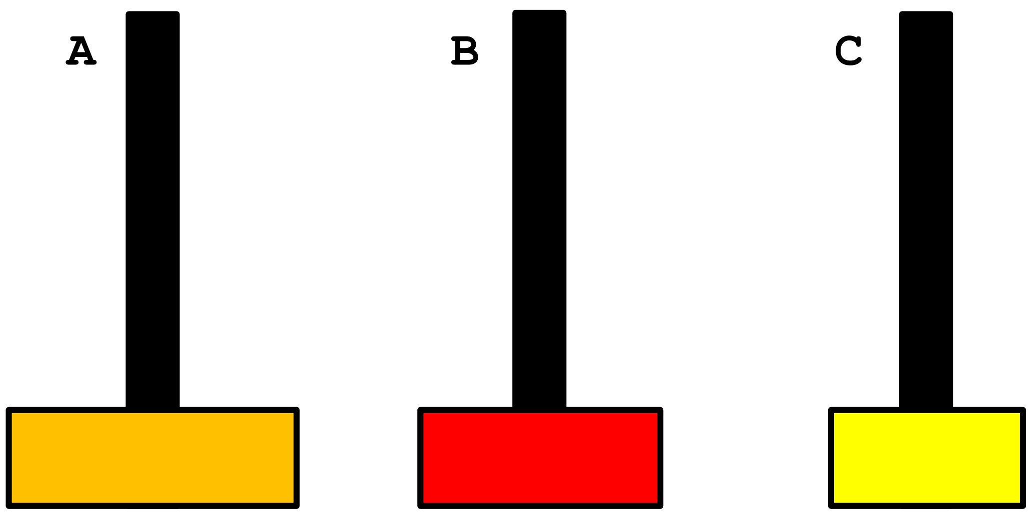
- n = 3



- move disk from A to B

Towers of Hanoi

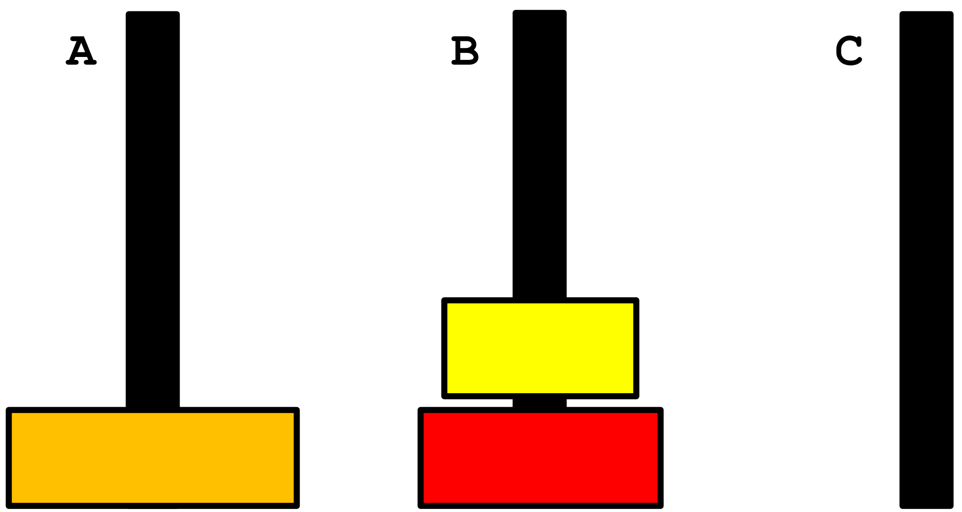
- n = 3



- move disk from C to B

Towers of Hanoi

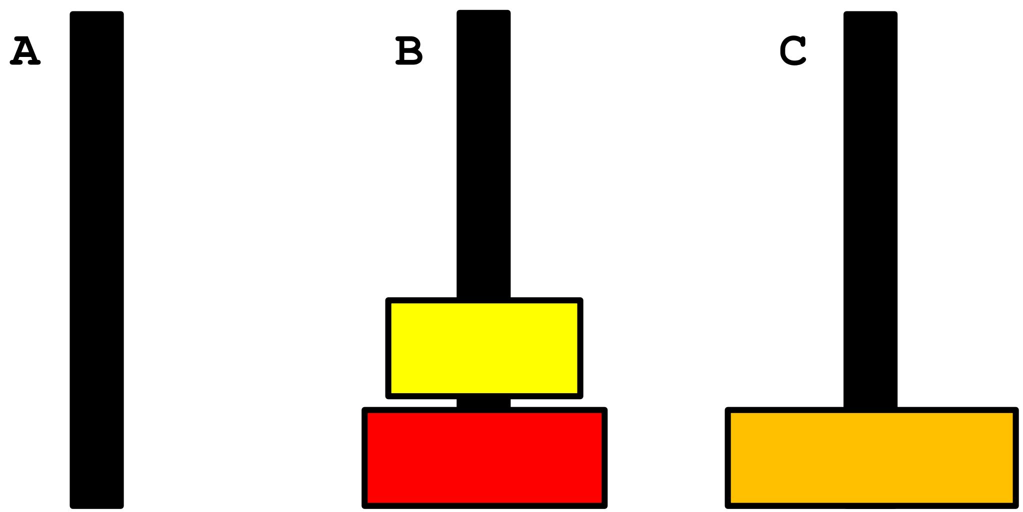
- n = 3



- move disk from A to C

Towers of Hanoi

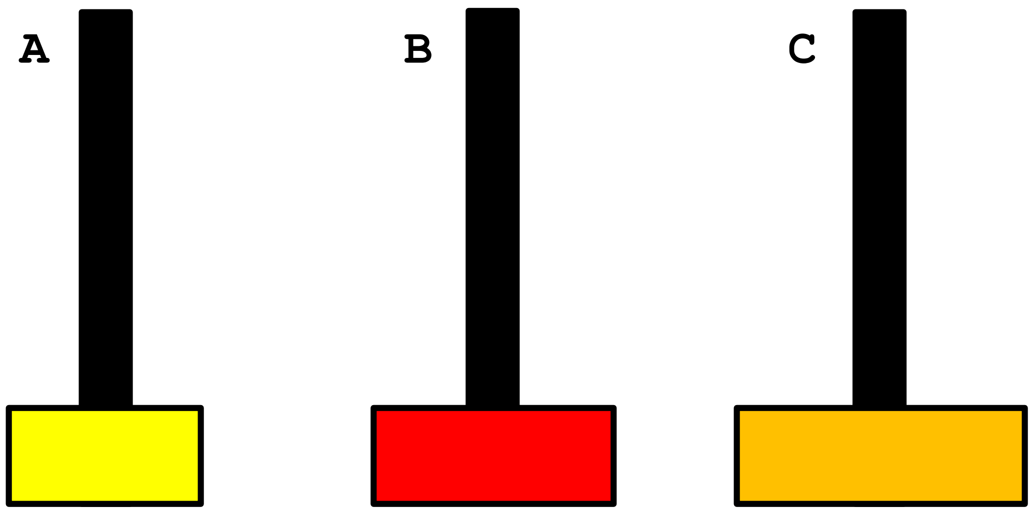
- n = 3



- move disk from B to A

Towers of Hanoi

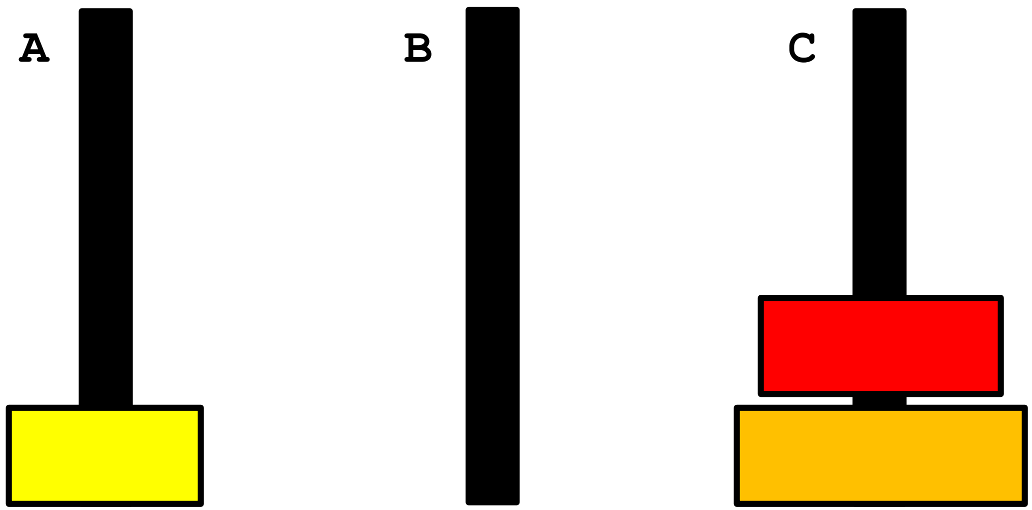
- n = 3



- move disk from B to C

Towers of Hanoi

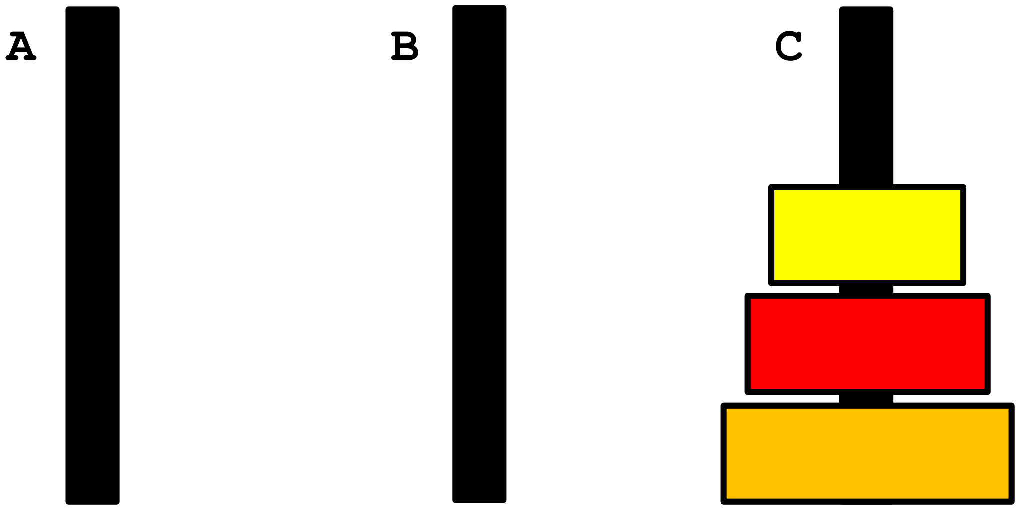
- n = 3



- move disk from A to C

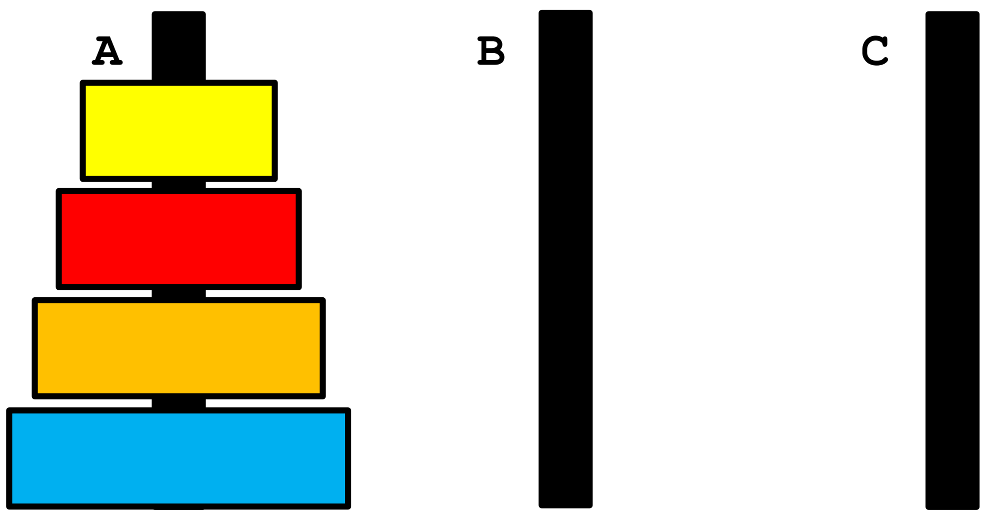
Towers of Hanoi

- n = 3



Towers of Hanoi

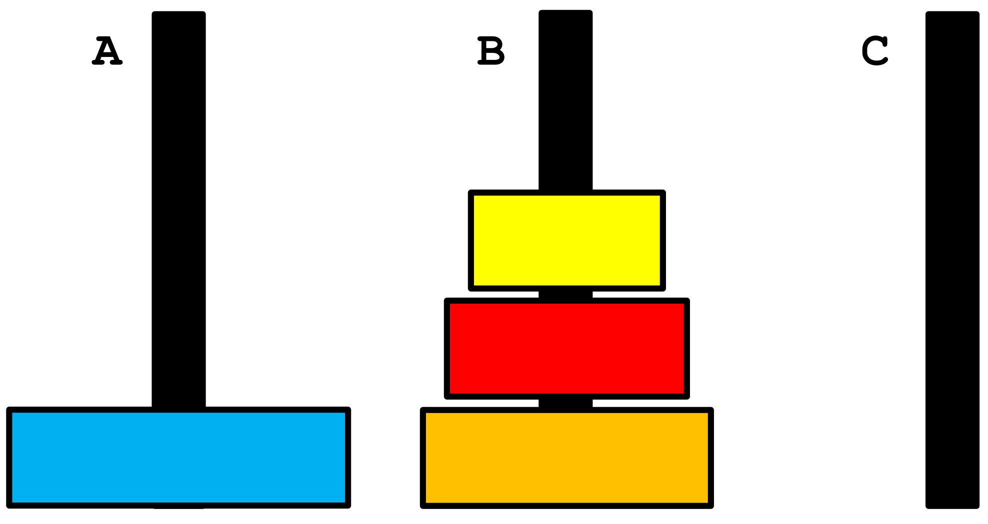
- n = 4



- move (n – 1) disks from A to B using C

Towers of Hanoi

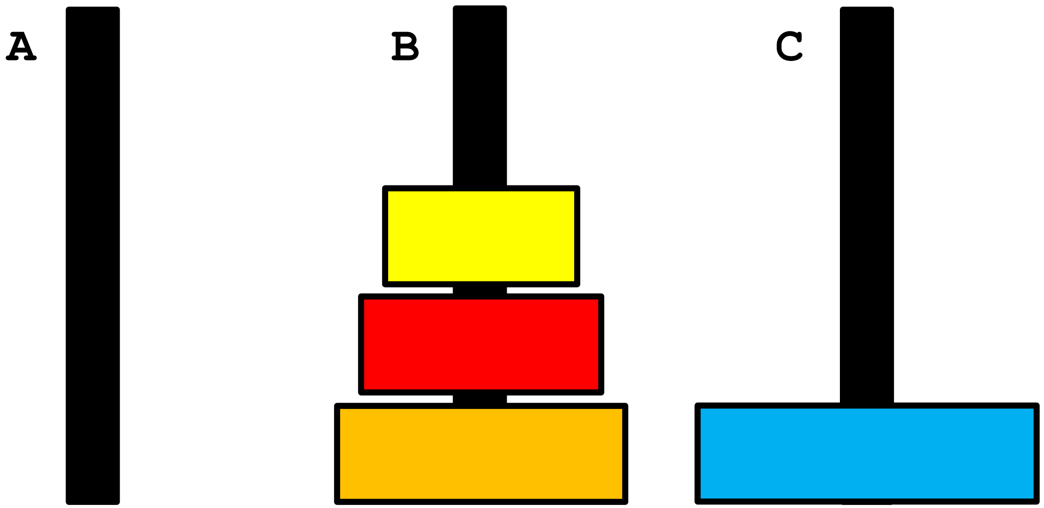
- n = 4



- move disk from A to C

Towers of Hanoi

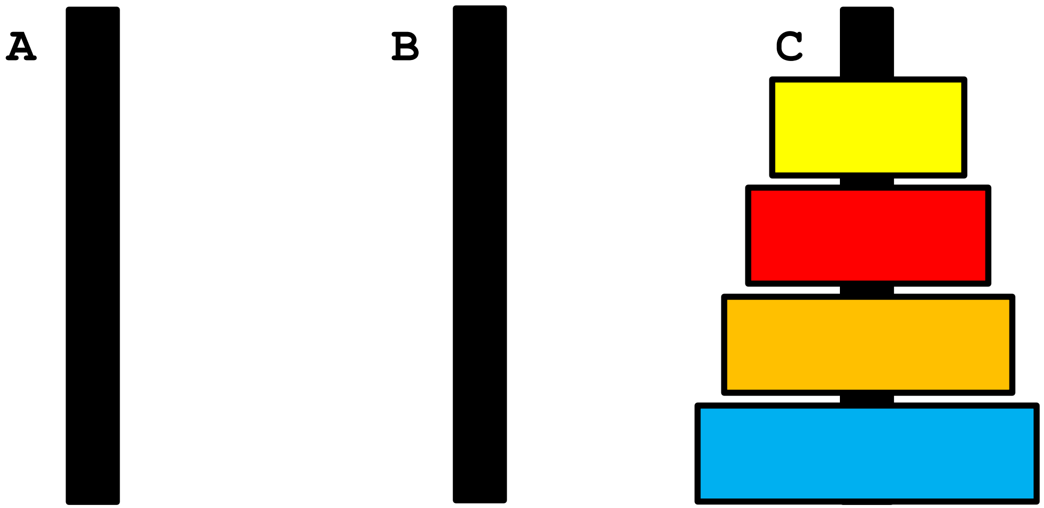
- n = 4



- move (n – 1) disks from B to C using A

Towers of Hanoi

- n = 4



- base case $n = 1$
  1. move disk from A to C
- recursive case
  1. move $(n - 1)$ disks from A to B
  2. move 1 disk from A to C
  3. move $(n - 1)$ disks from B to C

Towers of Hanoi

```
public static void move(int n,
                        String from,
                        String to,
                        String using) {
  if(n == 1) {
    System.out.println("move disk from " + from + " to " + to);
  }
  else {
    move(n - 1, from, using, to);
    move(1, from, to, using);
    move(n - 1, using, to, from);
  }
}
```