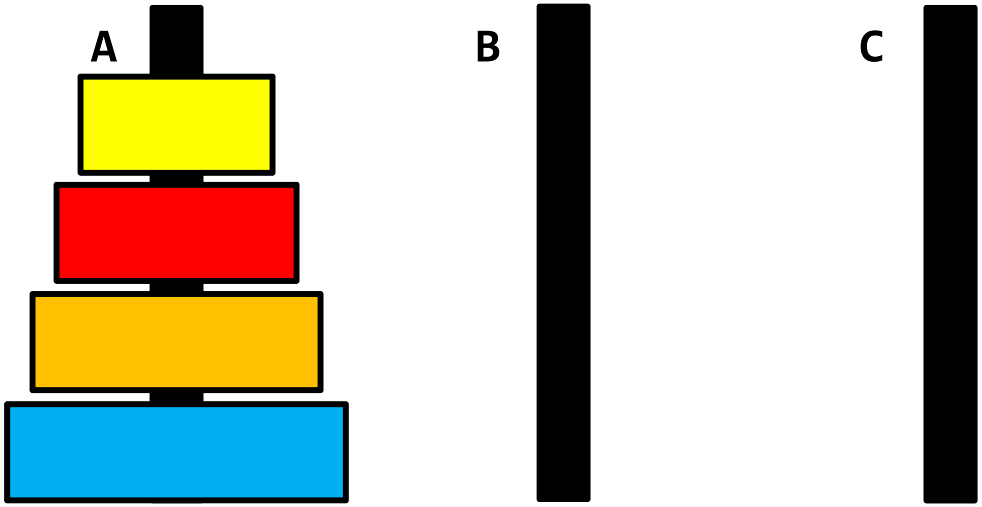
Recursion

notes Chapter 8

Tower of Hanoi



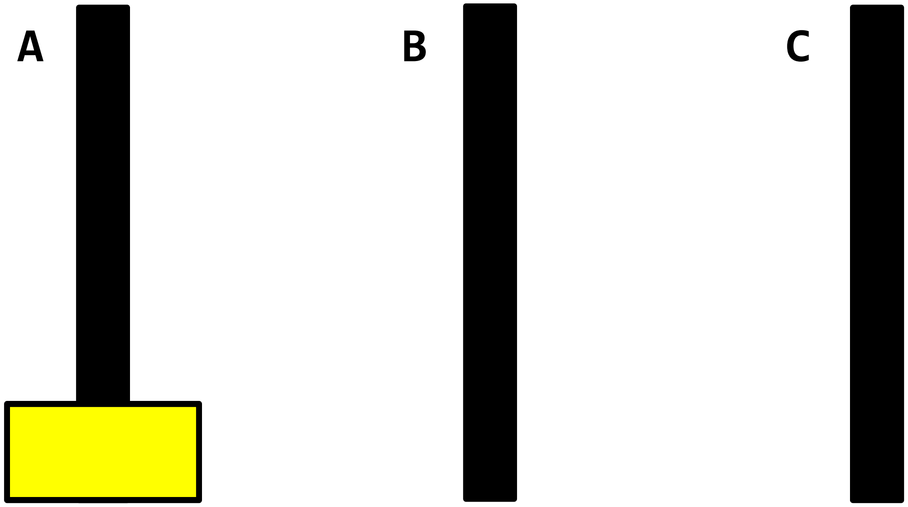
- move the stack of *n* disks from A to C
  - can move one disk at a time from the top of one stack onto another stack
  - cannot move a larger disk onto a smaller disk

Tower 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

Tower of Hanoi

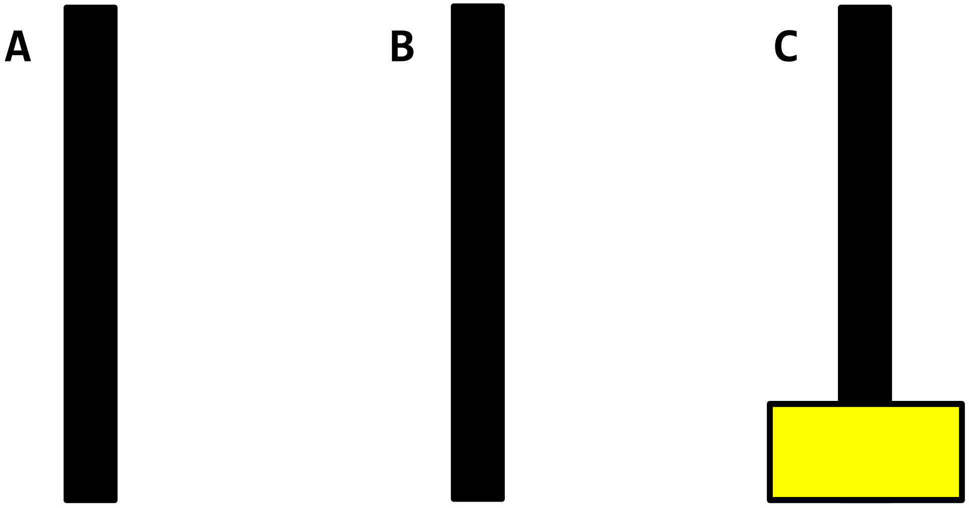
- n = 1



- move disk from A to C

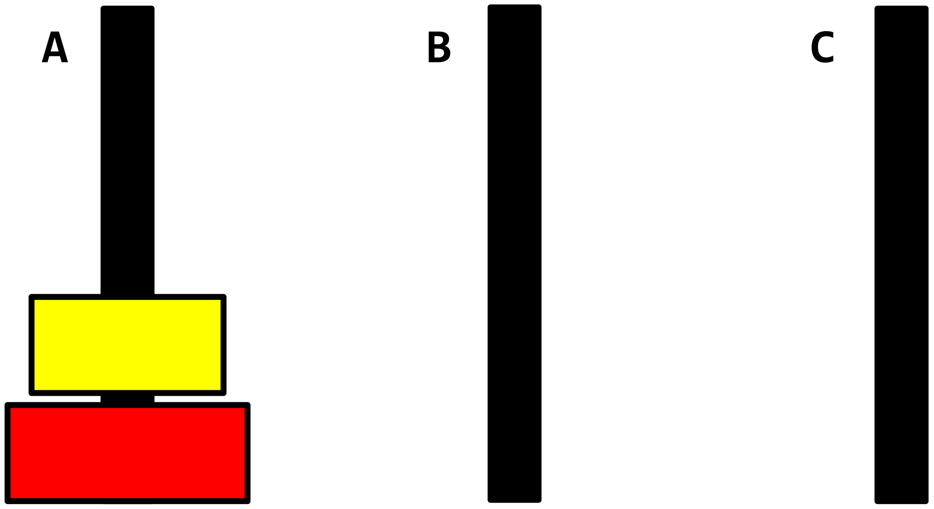
Tower of Hanoi

- n = 1



Tower of Hanoi

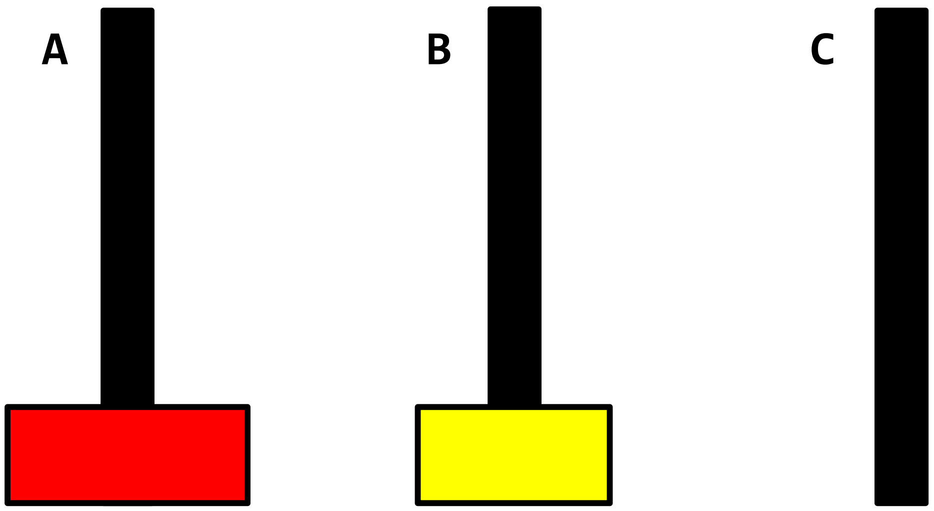
- n = 2



- move disk from A to B

Tower of Hanoi

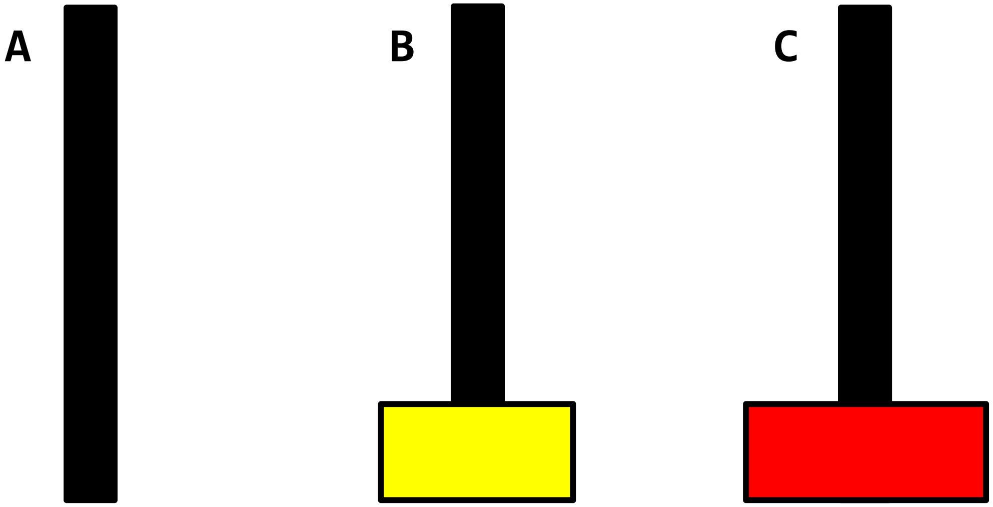
- n = 2



- move disk from A to C

Tower of Hanoi

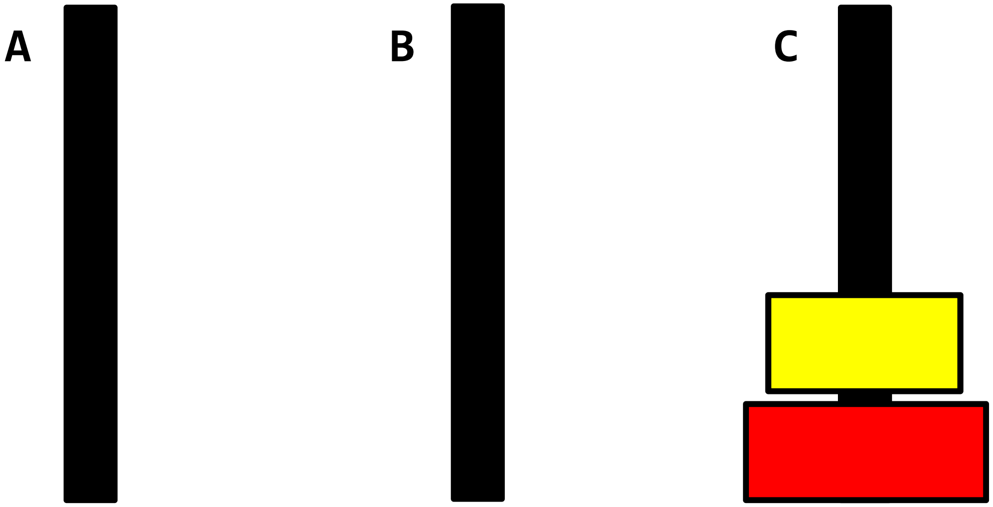
- n = 2



- move disk from B to C

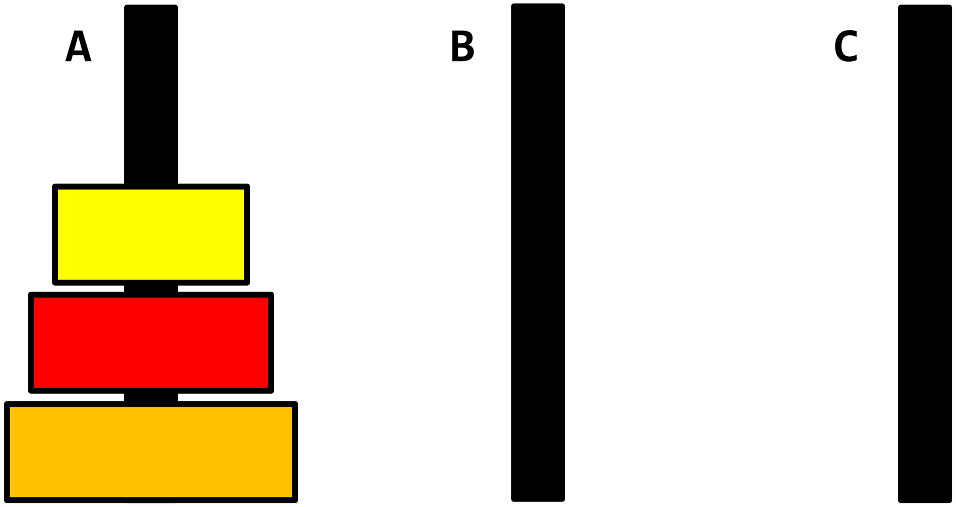
Tower of Hanoi

- n = 2



Tower of Hanoi

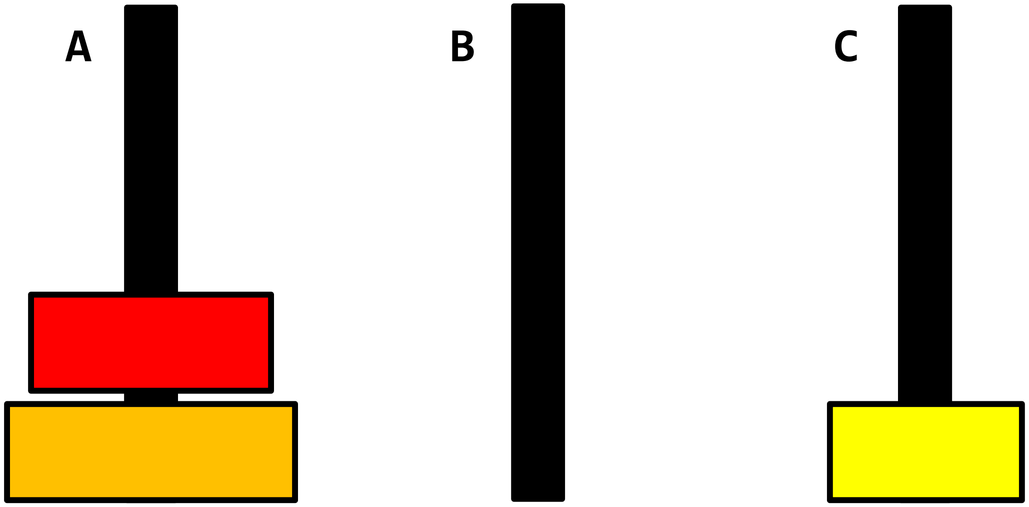
- n = 3



- move disk from A to C

Tower of Hanoi

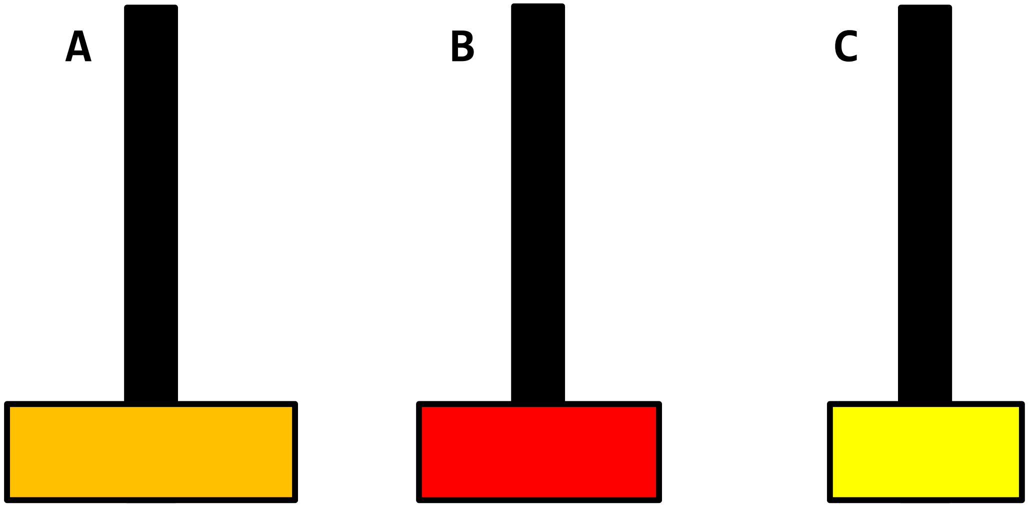
- n = 3



- move disk from A to B

Tower of Hanoi

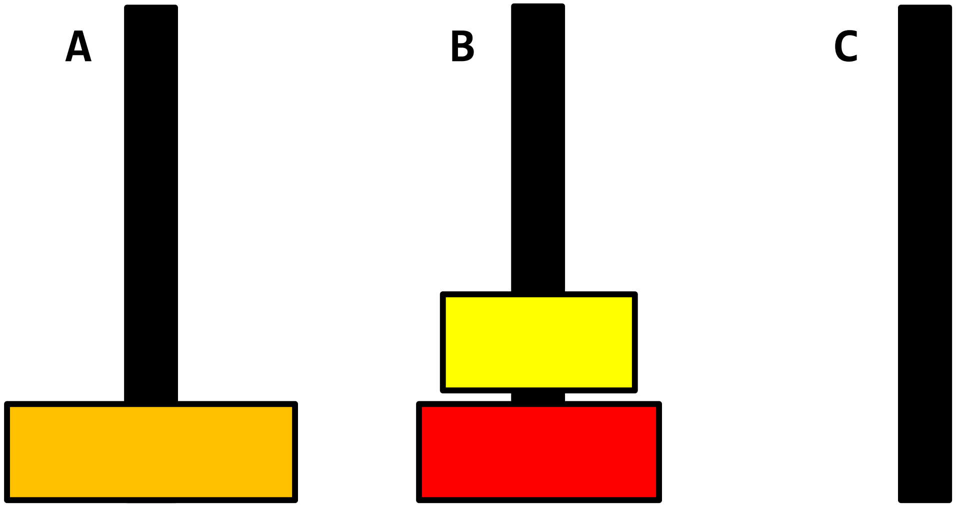
- n = 3



- move disk from C to B

Tower of Hanoi

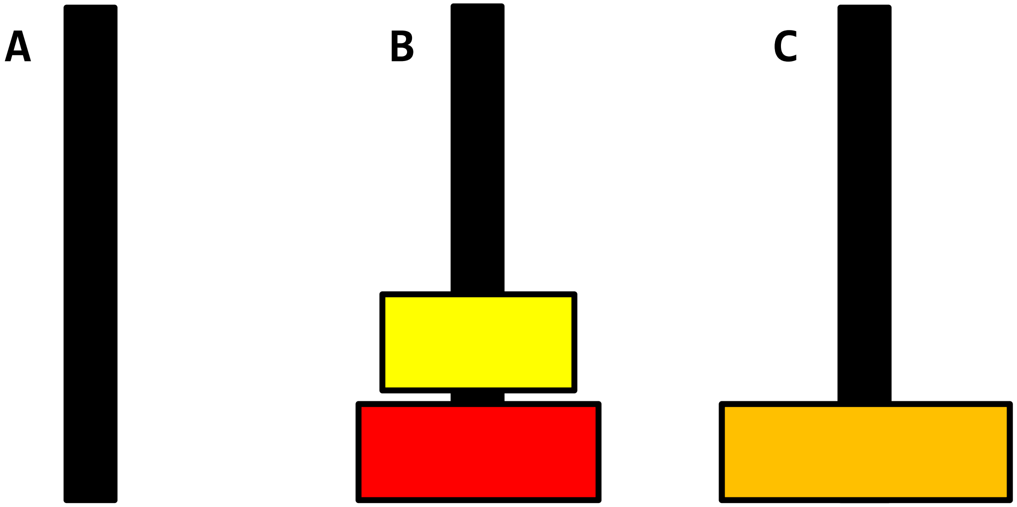
- n = 3



- move disk from A to C

Tower of Hanoi

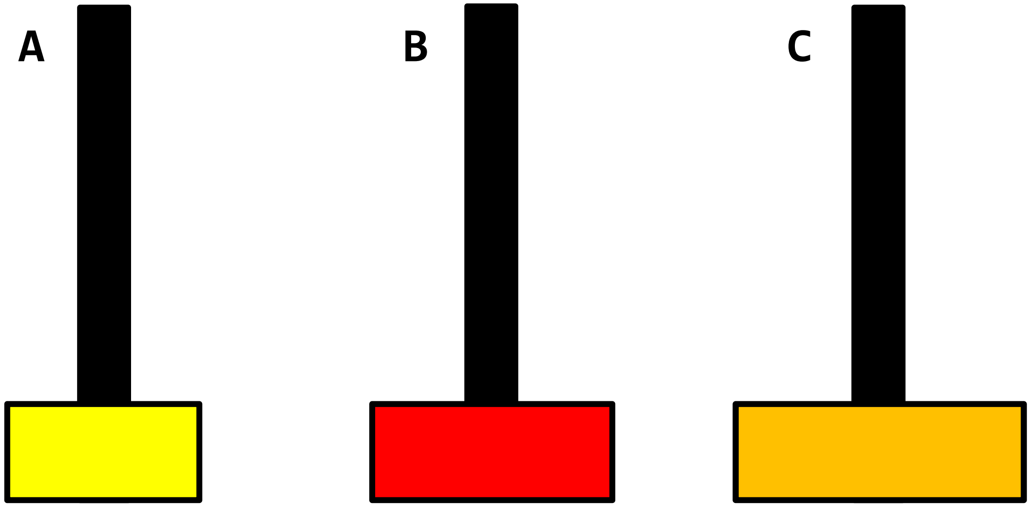
- n = 3



- move disk from B to A

Tower of Hanoi

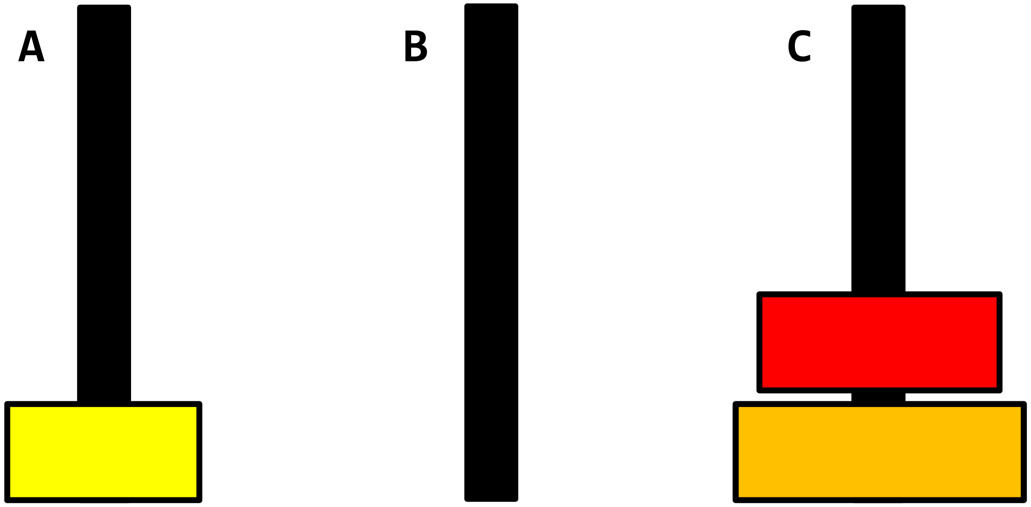
- n = 3



- move disk from B to C

Tower of Hanoi

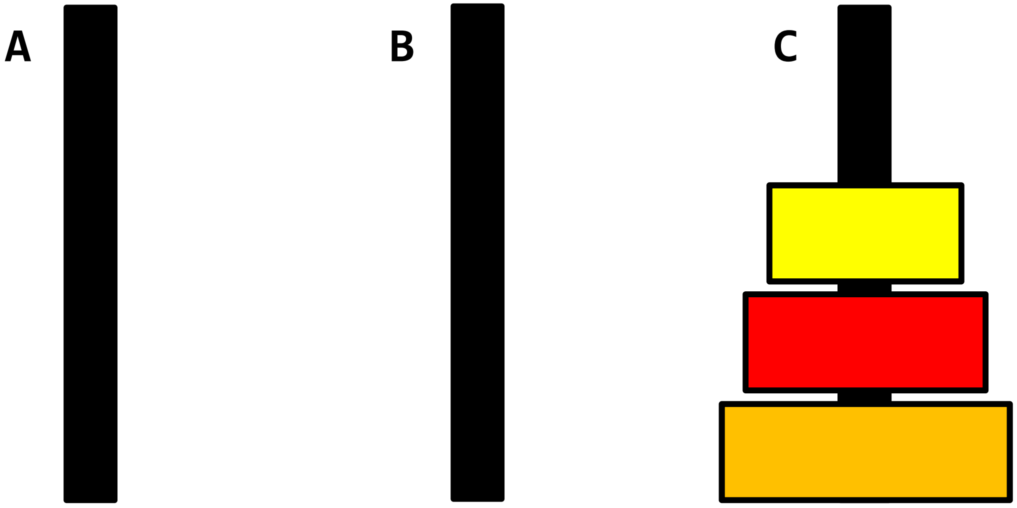
- n = 3



- move disk from A to C

Tower of Hanoi

- n = 3



Tower of Hanoi

- write a loop-based method to solve the Tower of Hanoi problem
  - discuss amongst yourselves now…

Tower of Hanoi

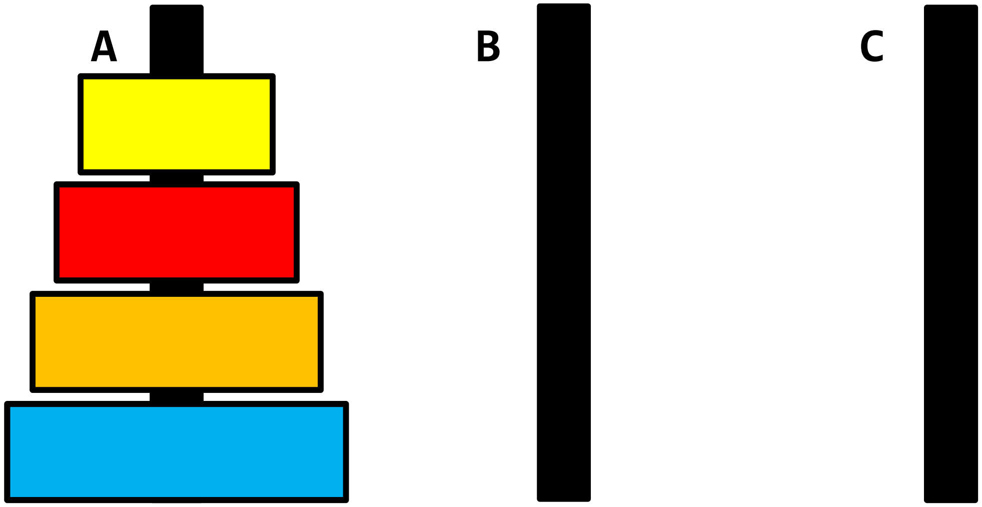
- imagine that you had the following method (see next slide)
    - how would you use the method to solve the Tower of Hanoi problem?
        - discuss amongst yourselves now…

Tower of Hanoi

```
/**
 * Prints the sequence of moves required to move n disks from the
 * starting pole (from) to the goal pole (to) using a third pole
 * (using).
 *
 * @param n
 *           the number of disks to move
 * @param from
 *           the starting pole
 * @param to
 *           the goal pole
 * @param using
 *           a third pole
 * @pre. n is greater than 0
 */
public static void move(int n, String from, String to, String using)
```

Tower of Hanoi

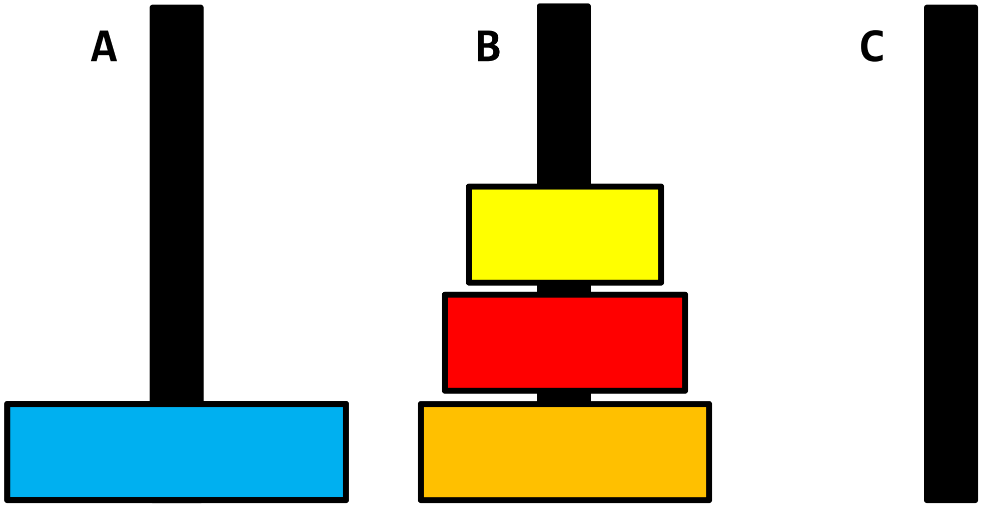
- n = 4



- you eventually end up at (see next slide)…

Tower of Hanoi

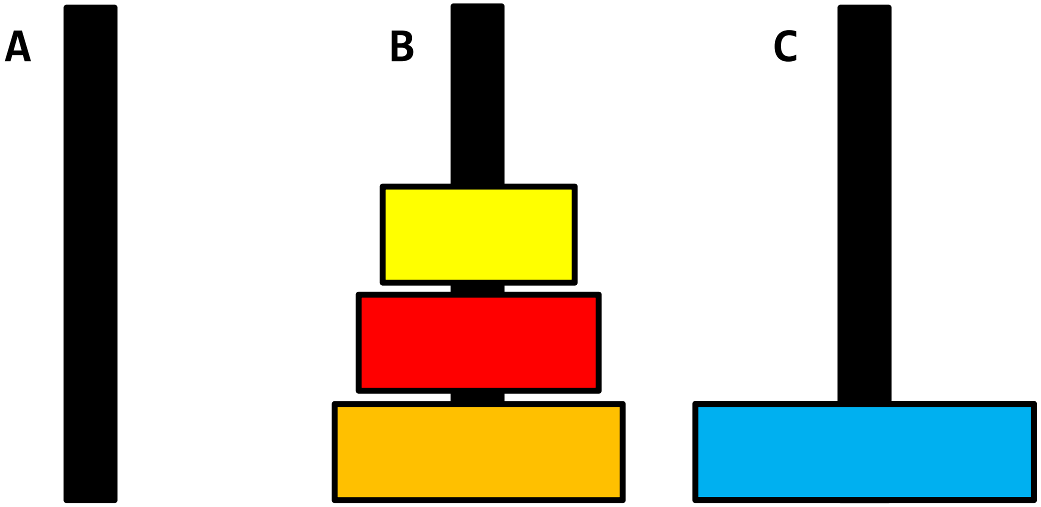
- n = 4



- move disk from A to C

Tower of Hanoi

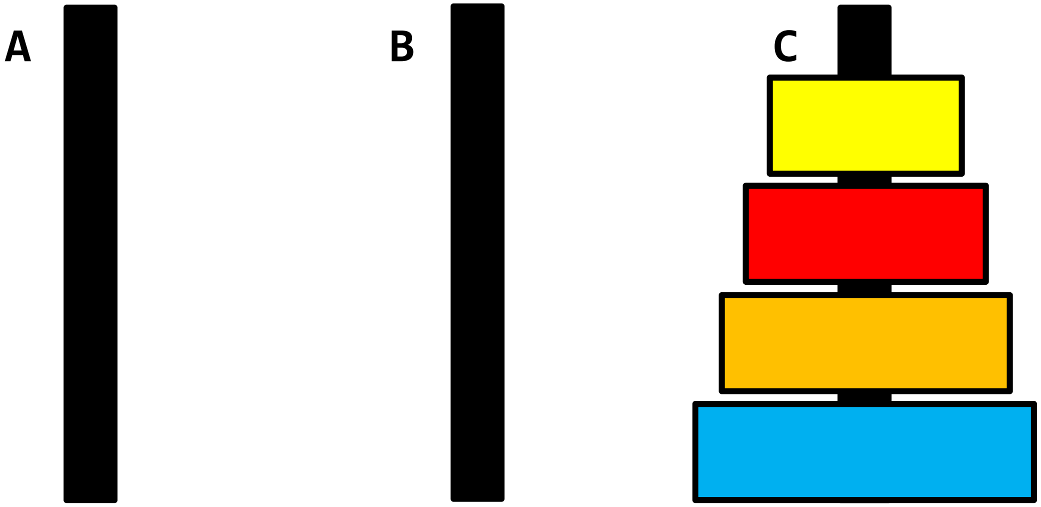
- n = 4



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

Tower of Hanoi

- $n = 4$



Tower of Hanoi

- notice that to solve the n = 4 size problem, you have to solve a variation of the n = 3 size problem twice and a variation of the n = 1 size problem once
- we can use the **move** method to solve these 3 sub-problems

Tower of Hanoi

- the basic solution can be described as follows:
  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

- furthermore:
  - if exactly n == 1 disk is moved, print out the starting pole and the goal pole for the move

Tower 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 3 disks from A to B
    move(1, from, to, using);         – move 1 disk from A to C
    move(n - 1, using, to, from);     – move 3 disks from B to C
  }
}
```

Printing n of Something

- suppose you want to implement a method that prints out n copies of a string

```
public static void printIt(String s, int n) {
  for(int i = 0; i < n; i++) {
    System.out.print(s);
  }
}
```

A Different Solution

- alternatively we can use the following algorithm:
  1. if n == 0 done, otherwise
     I. print the string once
     II. print the string (n – 1) more times

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);    // method invokes itself
  }
}
```

print s once

print s (n-1) times

Recursion

- a method that calls itself is called a *recursive* method
- a recursive method solves a problem by ~~repeatedly~~ reducing the problem so that a base case can be reached

```
printItToo("*", 5)
*printItToo ("*", 4)
**printItToo ("*", 3)
***printItToo ("*", 2)
****printItToo ("*", 1)
*****printItToo ("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Infinite Recursion

- if the base case(s) is missing, or never reached, a recursive method will run forever (or until the computer runs out of resources)

```
public static void printItForever(String s, int n) {
  // missing base case; infinite recursion
  System.out.print(s);
  printItForever(s, n - 1);
}
```

```
 printItForever("*", 1)
 * printItForever("*", 0)
 ** printItForever("*", -1)
 *** printItForever("*", -2) ...........
```

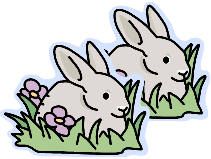
leads to Stack Over flow Exception

Climbing a Flight of n Stairs

- not Java

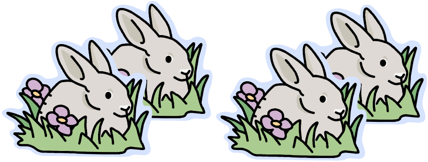
```
/**
  * method to climb n stairs
  */
climb(n) :
if n == 0
  done
else
  step up 1 stair
  climb(n – 1);
end
```

Rabbits



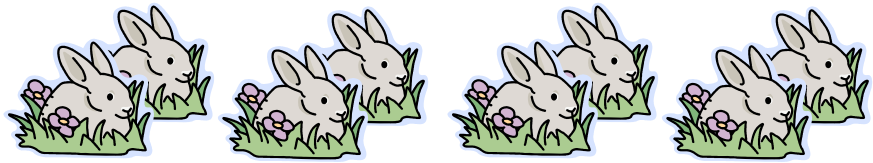
Month 0: 1 pair

0 additional pairs



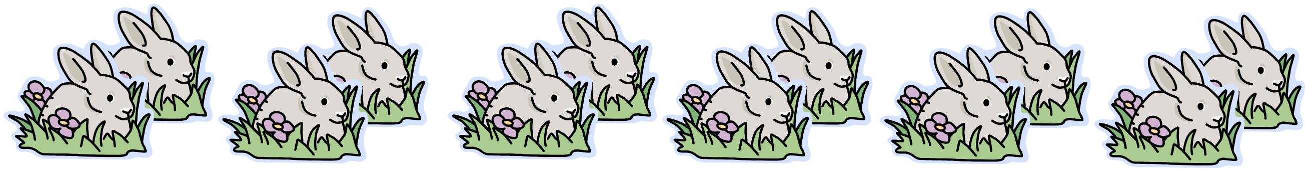
Month 1: first pair makes another pair

1 additional pair



Month 2: each pair makes another pair; oldest pair dies

1 additional pair



2 additional pairs

Month 3: each pair makes another pair; oldest pair dies

Fibonacci Numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n - 1) +  F(n - 2)`

Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

base case $F(0) = 0$

base case $F(1) = 1$

recursive case $F(n) = F(n-1) + F(n-2)$

$n-1 < n$

$n-2 < n$

Recursive Methods & Return Values

- write a recursive method that multiplies two positive integer values (i.e., both values are strictly greater than zero)

- observation: $m \times n$ means add m n's together
  - in other words, you can view multiplication as recursive addition

Recursive Methods & Return Values

- not Java:

```
/**
 * Computes m * n
 */
multiply(m, n) :
if m == 1
    return n
else
    return n + multiply(m - 1, n)
```

$m-1 < m$

```
public static int multiply(int m, int n) {
  if (m == 1) {
    return n;
  }
  return n + multiply(m - 1, n);
}
```

base case $1 \times n$

recursive case

$n + (m-1) \times n$

Recursive Methods & Return Values

- example: write a recursive method **countZeros** that counts the number of zeros in an integer number **n**
  - **10305060700002L** has 8 zeros

loop over digits in the number ?

- trick: examine the following sequence of numbers

1. **10305060700002** not == 0 0 + countZeros ( )
2. **1030506070000** == 0 1 + countZeros ( )
3. **103050607000** == 0 1 + countZeros ( )
4. **10305060700** == 0 1 + countZeros ( )
5. **1030506070** == 0 1 + countZeros ( )
6. **103050607** ... not == 0 0 + countZeros ( )

Recursive Methods & Return Values

- not Java:

```
/**
 * Counts the number of zeros in an integer n
 */
countZeros(n) :
if the last digit in n is a zero
    return 1 + countZeros(n / 10)
else
    return countZeros(n / 10)
```

# of zeros in the number after removing the last digit

- don't forget to establish the base case(s)
  - when should the recursion stop? when you reach a single digit (not zero digits; you never reach zero digits!)
    - base case #1 : `n == 0`
      - □ `return 1`
    - base case #2 : `n != 0 && n < 10`
      - □ `return 0`

```
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

base cases

n is 0

n is 1, 2, 3, 4, 5, 6, 7, 8, or 9

recursive case

is the remainder after dividing by 10 equal to 0?

countZeros Call Stack

`countZeros( 800410L )`

long

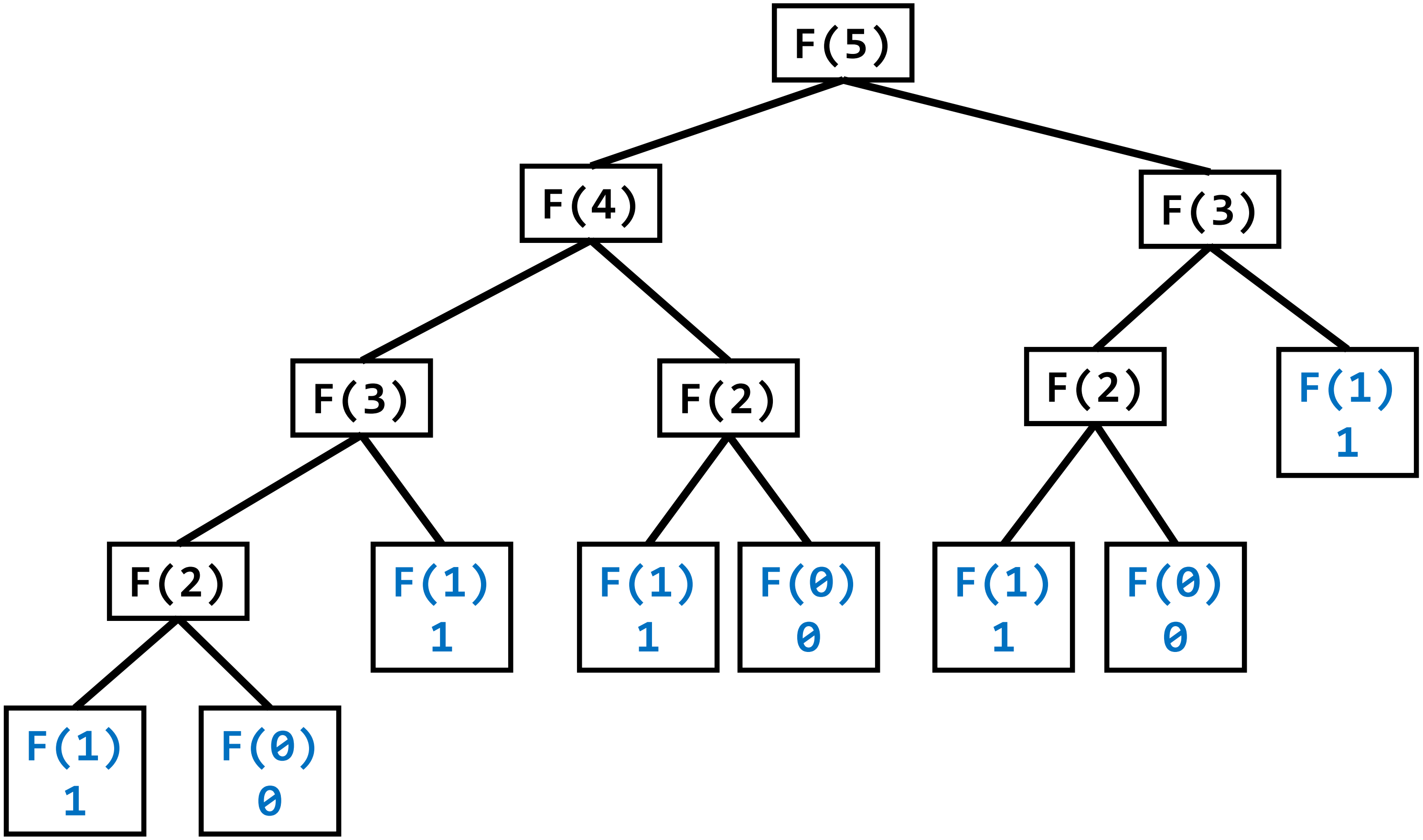
every instance of a method runs in its own chunk of memory

last in first out

`countZeros( 8L )`	`0`	
`countZeros( 80L )`	`1 + 0`	the number of zeros in 8
`countZeros( 800L )`	`1 + 1 + 0`	the number of zeros in 80
`countZeros( 8004L )`	`0 + 1 + 1 + 0`	the number of zeros in 800
`countZeros( 80041L )`	`0 + 0 + 1 + 1 + 0`	the number of zeros in 8004
`countZeros( 800410L )`	`1 + 0 + 0 + 1 + 1 + 0`	the number of zeros in 80041

`= 3`

Fibonacci Call Tree



Compute Powers of 10

- write a recursive method that computes 10^n for any integer value **n**
- recall:
  - $10^0 = 1$ base case
  - $10^n = 10 * 10^{n-1}$
  - $10^{-n} = 1 / 10^n$

recursive cases

```
public static double powerOf10(int n) {
  if (n == 0) {
    // base case
    return 1.0;
  }
  else if (n > 0) {
    // recursive call for positive n
    return 10.0 * powerOf10(n - 1);
  }
  else {
    // recursive call for negative n
    return 1.0 / powerOf10(-n);
  }
}
```

$n-1 < n$

Fibonacci Numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n – 1) +  F(n – 2)`

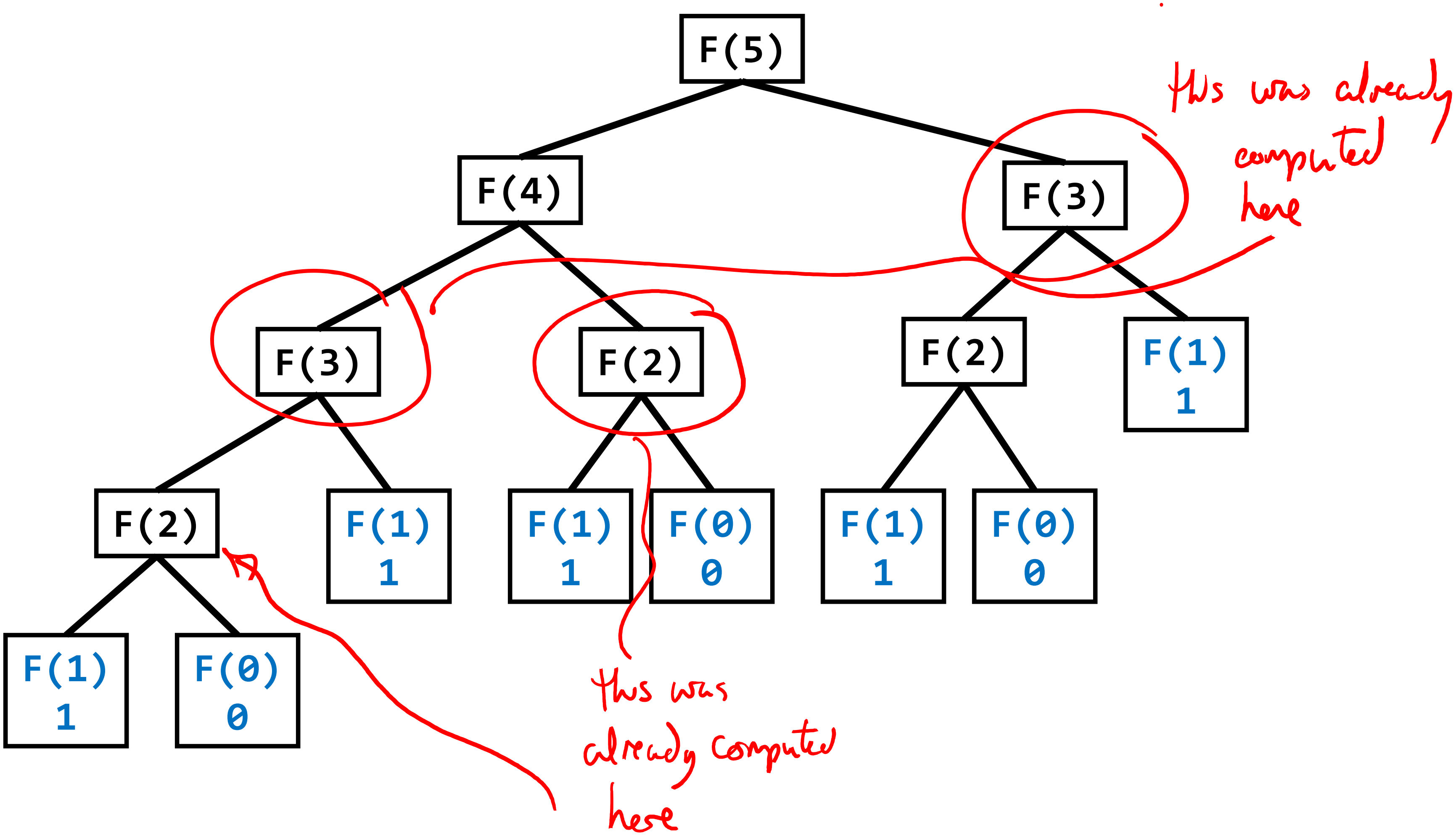
Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

Fibonacci Call Tree

$O(2^n)$



A Better Recursive Fibonacci

static initialization block

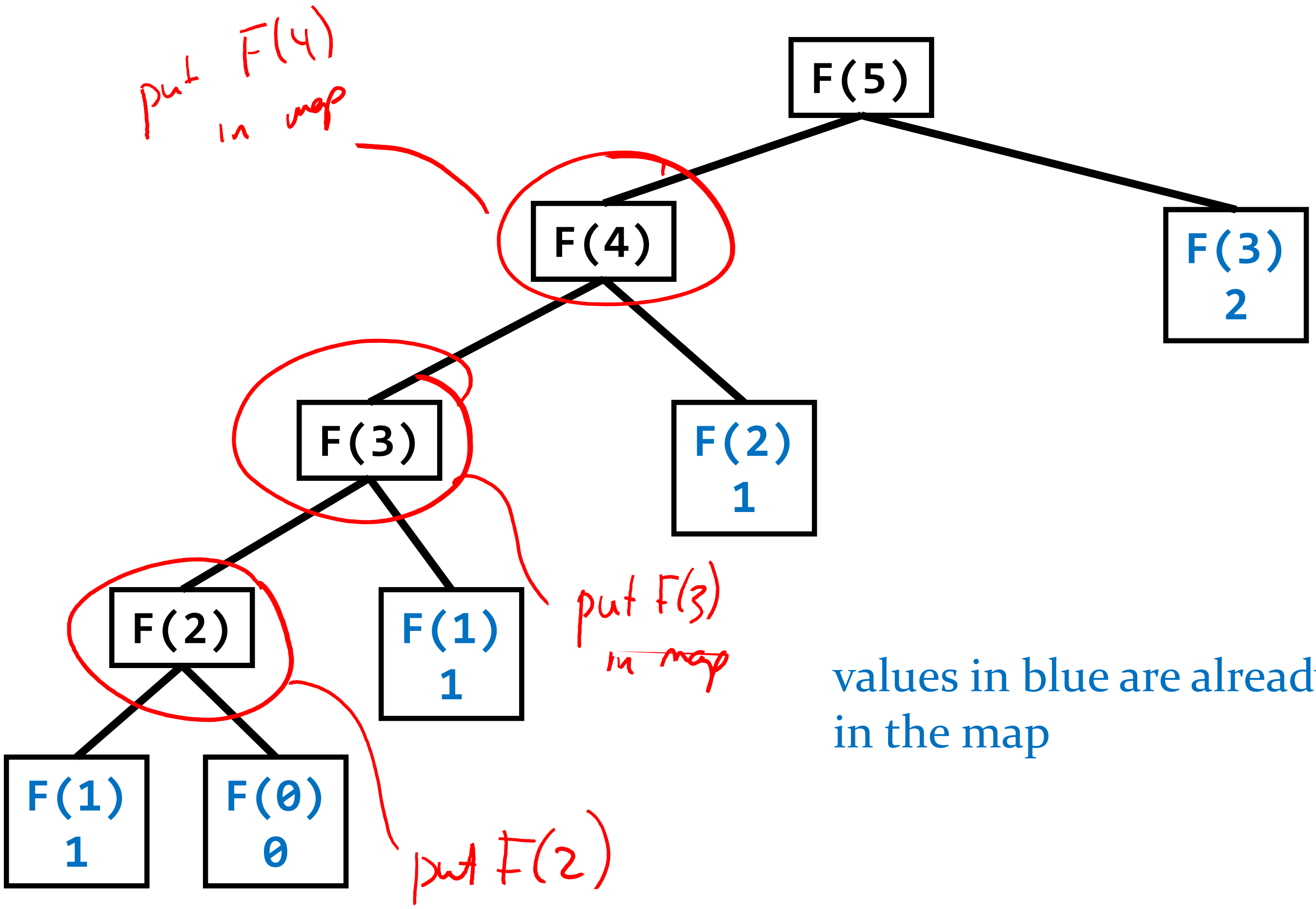
```
public class Fibonacci {
 private static Map<Integer, Long> values = new HashMap<Integer, Long>();
 static {
   Fibonacci.values.put(0, (long) 0);
   Fibonacci.values.put(1, (long) 1);
 }

 public static long getValue(int n) {
   Long value = Fibonacci.values.get(n);
   if (value != null) {
     return value;
   }
   value = Fibonacci.getValue(n - 1) + Fibonacci.getValue(n - 2);
   Fibonacci.values.put(n, value);
   return value;
 }
}
```

key – n	F(n) – value
0	0
1	1
2	1
3	2
	⋮

look in the map for F(n)

put F(n) in the map so that we can re-use it when needed

Better Fibonacci Call Tree



put F(4) in map

put F(3) in map

put F(2) into the map

O(n)

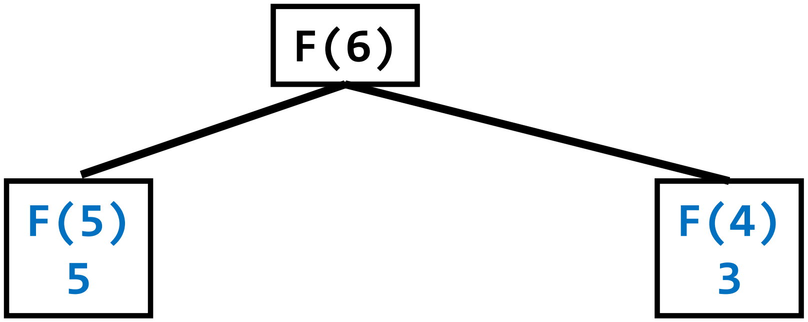
values in blue are already stored in the map

A Better Recursive Fibonacci

- because the map is static subsequent calls to **Fibonacci.*getValue*(int)** can use the values already computed and stored in the map

Better Fibonacci Call Tree

- assuming the client has already invoked **Fibonacci.*getValue*(5)**



values in blue are already stored in the map

Compute Powers of 10

- write a recursive method that computes **10^n** for any integer value **n**
- recall:
  - $10^n = 1 / 10^{-n}$ if $n < 0$
  - $10^0 = 1$
  - $10^n = 10 * 10^{n-1}$

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

A Better Powers of 10

- recall:
  - `10`^{n} `= 1 / 10`$^{-n}$ if `n < 0`
  - `10`^{0} `= 1`
  - `10`^{n} `= 10 * 10`$^{n-1}$ if **`n`** is odd
  - `10`^{n} `= 10`$^{n/2}$ `* 10`$^{n/2}$ if **`n`** is even

```
public static double powerOf10(int n) {
  if (n < 0) {
    return 1.0 / powerOf10(-n);
  }
  else if (n == 0) {
    return 1.0;
  }
  else if (n % 2 == 1) {          if n is odd
    return 10 * powerOf10(n - 1);
  }
  double value = powerOf10(n / 2);          else n is even
  return value * value;
}
```

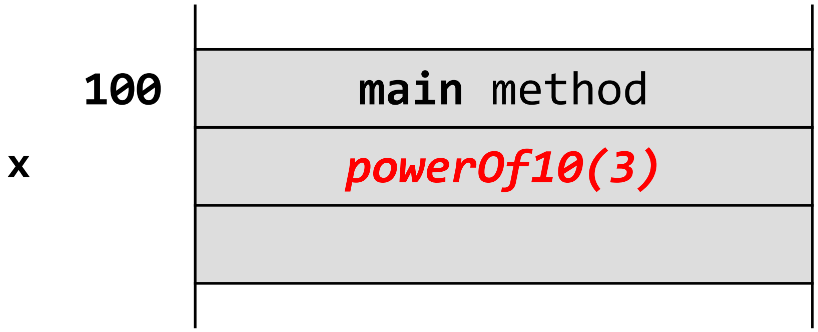
What happens during recursion

What Happens During Recursion

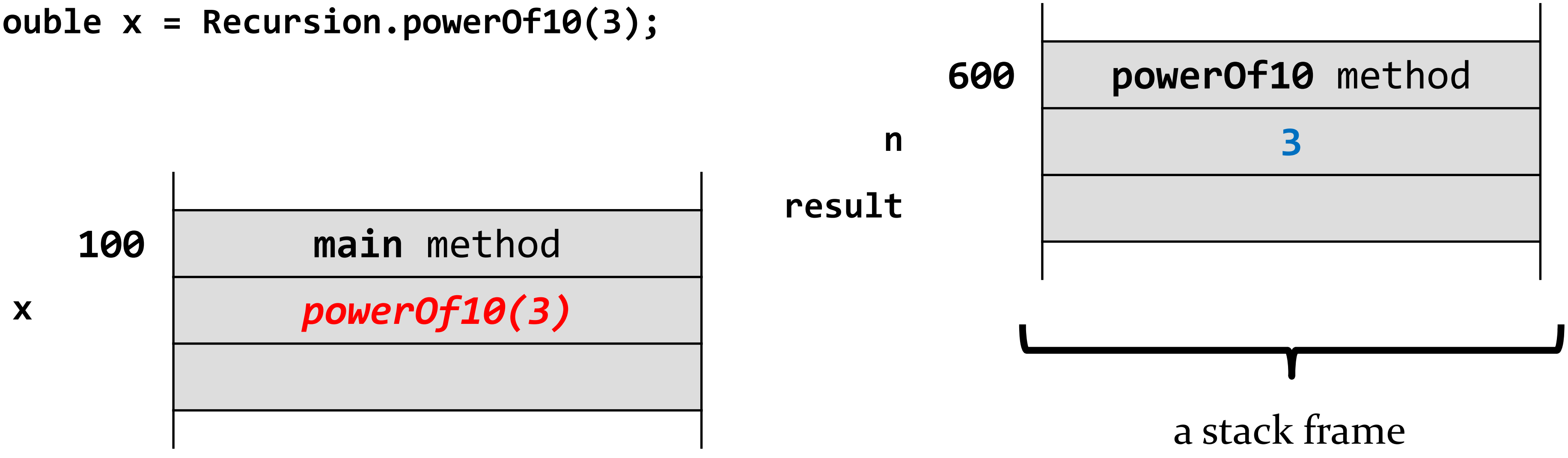
- a simplified model of what happens during a recursive method invocation is the following:
  - whenever a method is invoked that method runs in a *new* block of memory
    - when a method recursively invokes itself, a new block of memory is allocated for the newly invoked method to run in
- consider a slightly modified version of the **powerOf10** method

```
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);
```

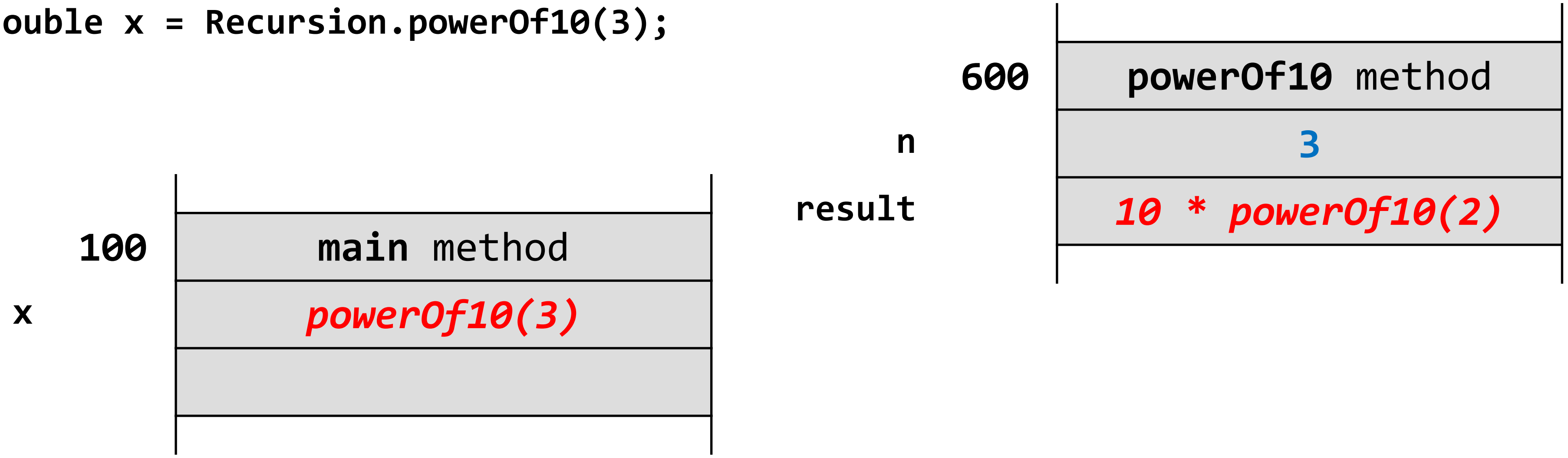


```
double x = Recursion.powerOf10(3);
```



- 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);
```



100 **main** method

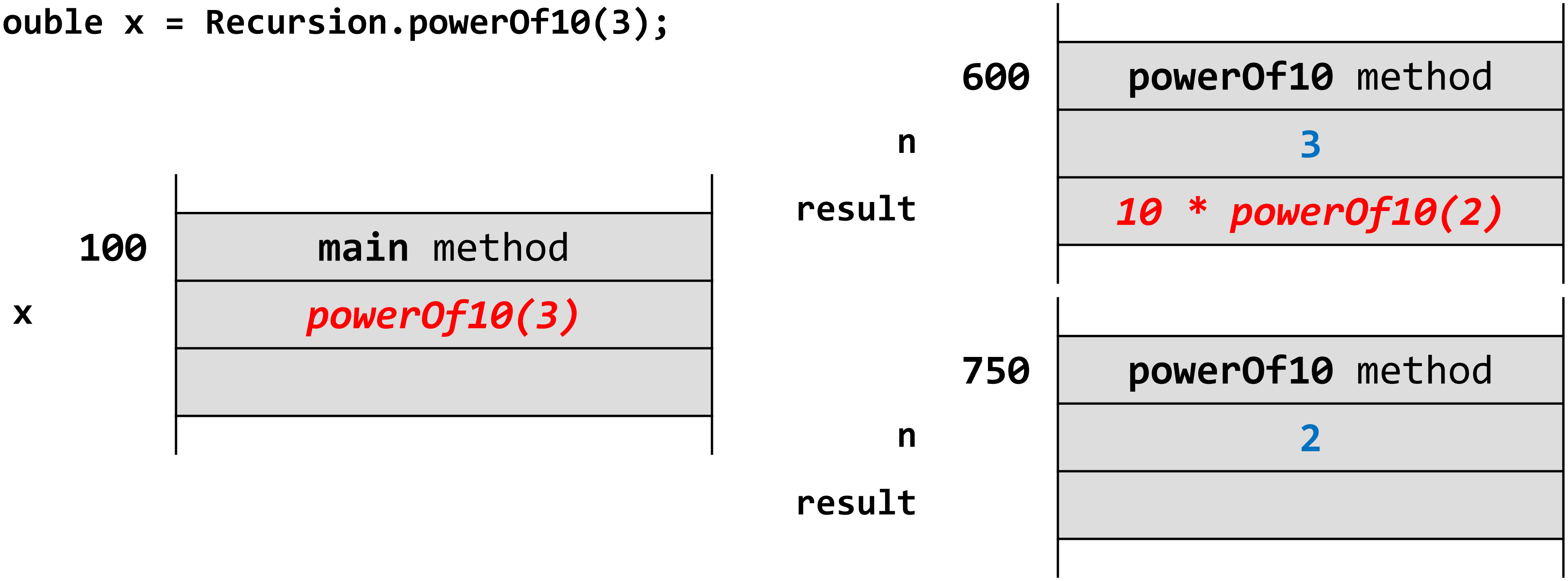
x *powerOf10(3)*

600 **powerOf10** method

n 3

result *10 * powerOf10(2)*

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

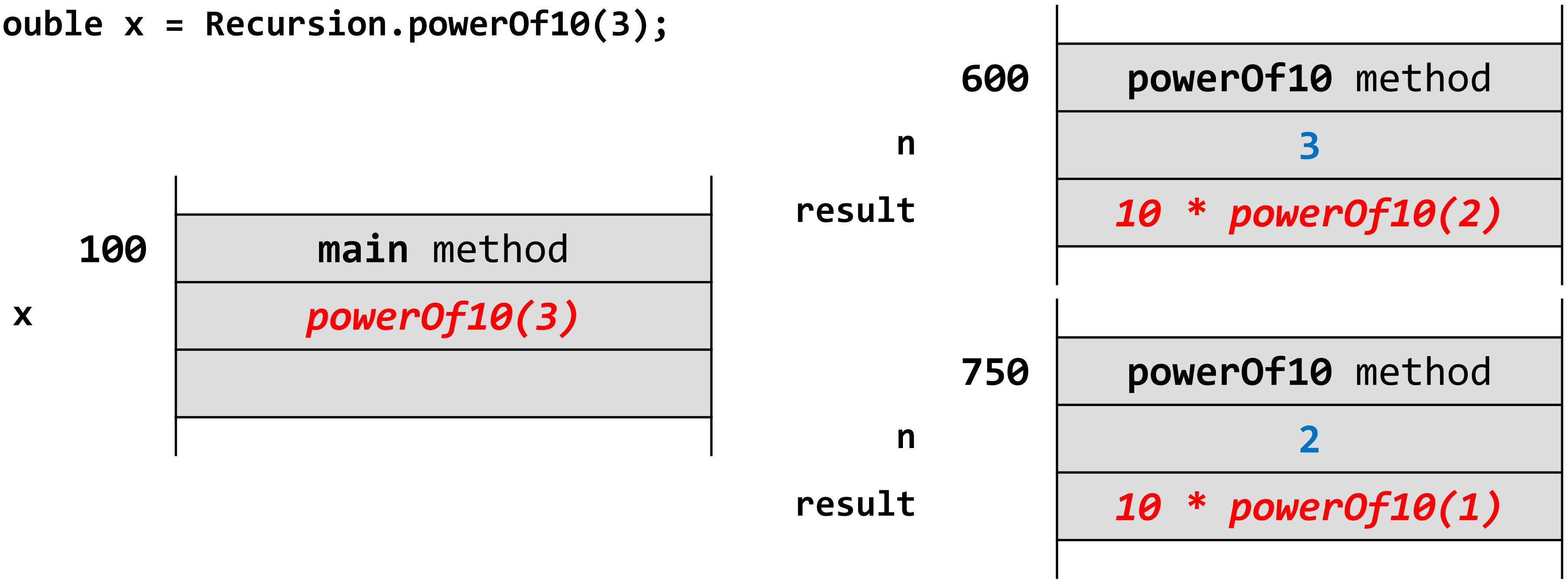
result 10 * powerOf10(2)

750 powerOf10 method

n 2

result

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

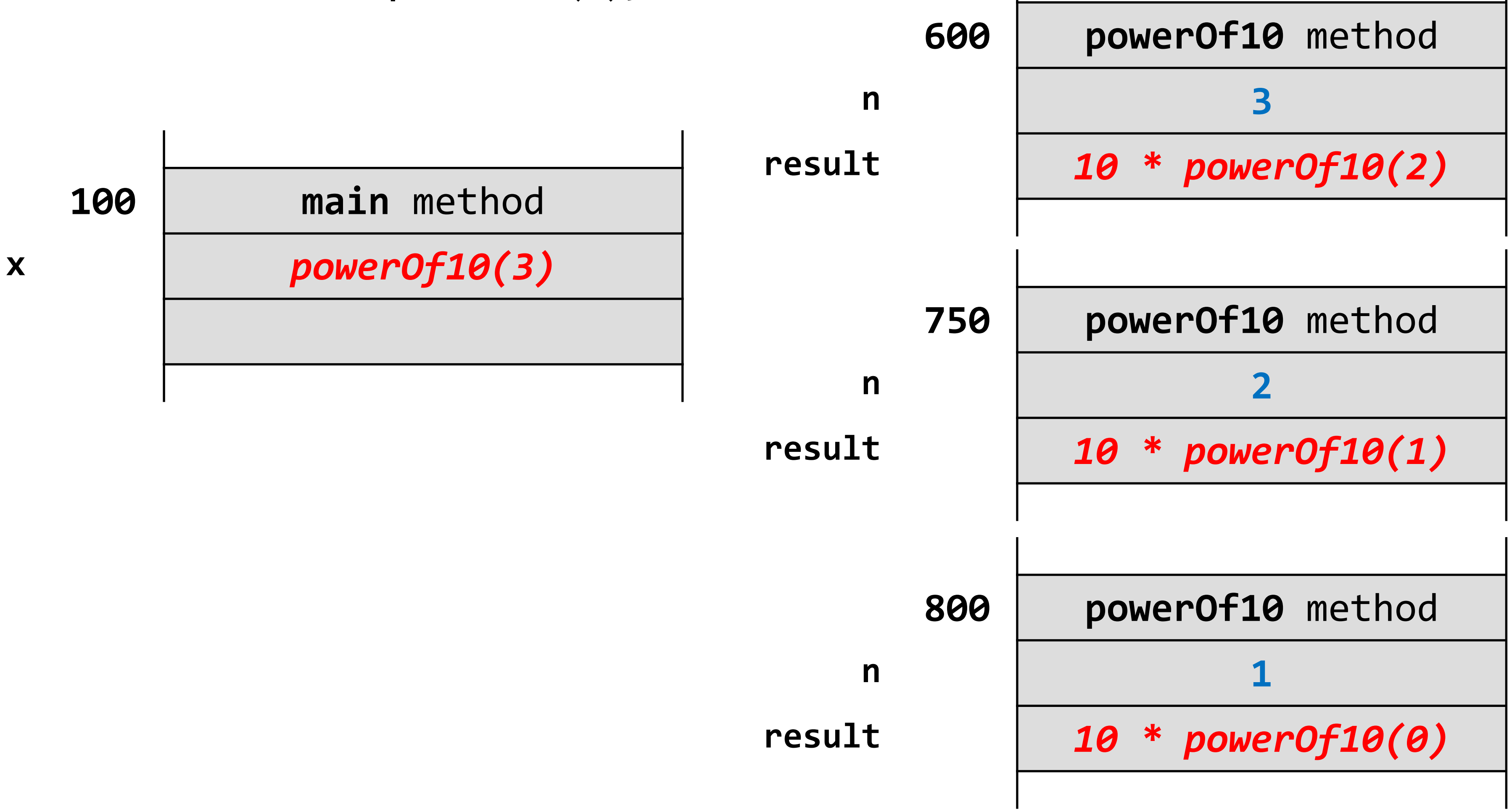
result 10 * powerOf10(2)

750 powerOf10 method

n 2

result 10 * powerOf10(1)

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

result 10 * powerOf10(2)

750 powerOf10 method

n 2

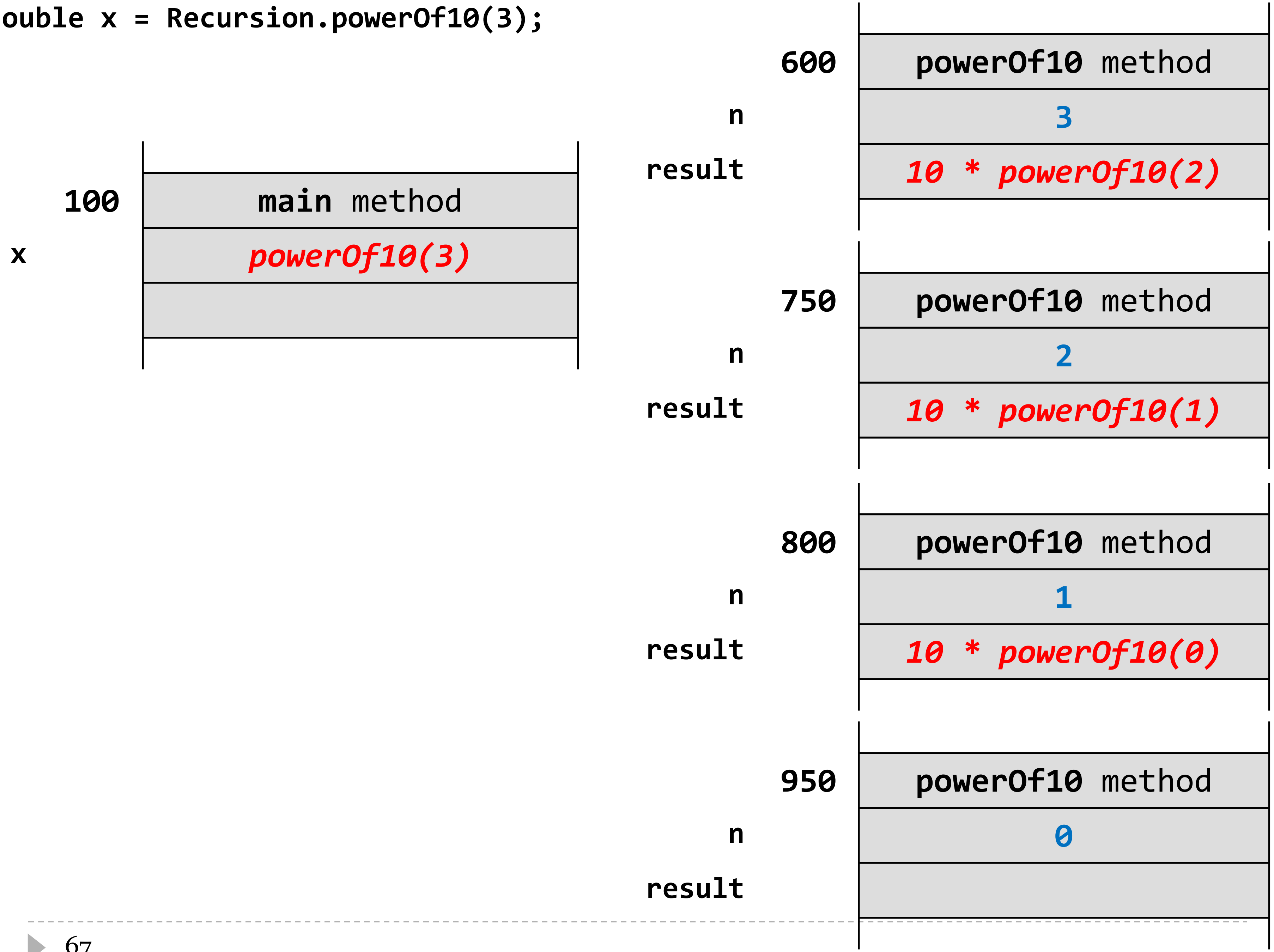
result 10 * powerOf10(1)

800 powerOf10 method

n 1

result 10 * powerOf10(0)

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



100

main method

x

powerOf10(3)

600

powerOf10 method

n

3

result

10 * powerOf10(2)

750

powerOf10 method

n

2

result

10 * powerOf10(1)

800

powerOf10 method

n

1

result

10 * powerOf10(0)

950

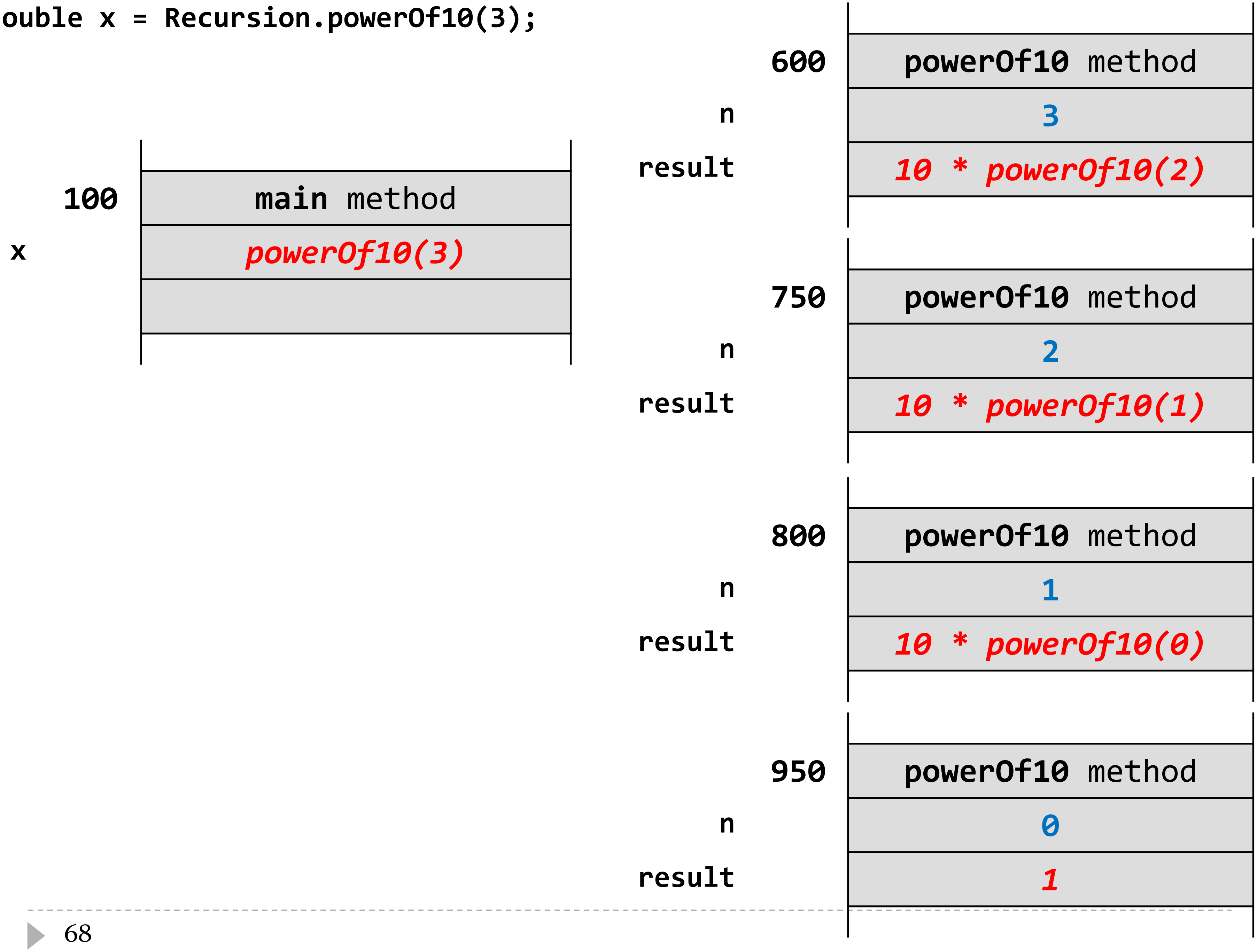
powerOf10 method

n

0

result

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

result 10 * powerOf10(2)

750 powerOf10 method

n 2

result 10 * powerOf10(1)

800 powerOf10 method

n 1

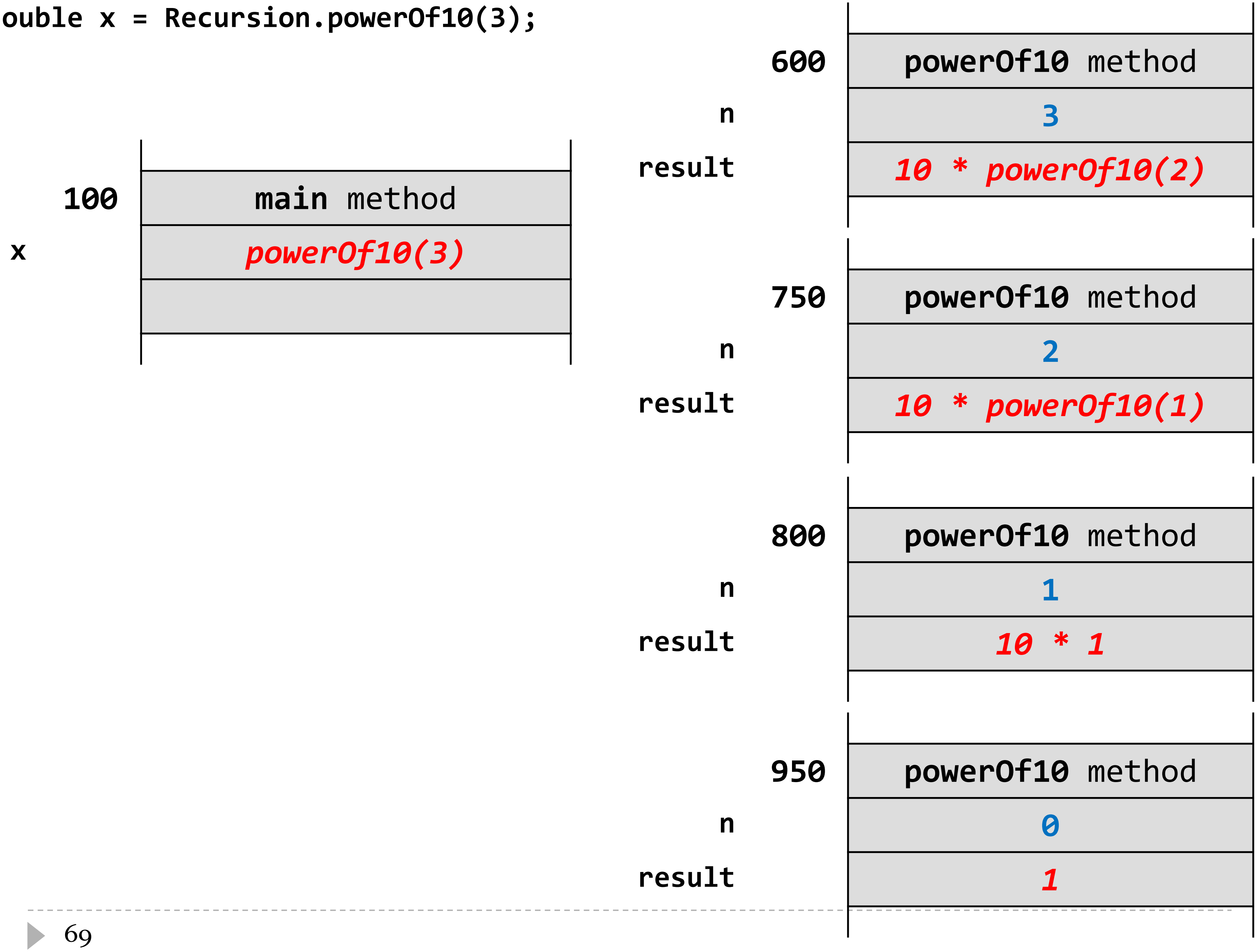
result 10 * powerOf10(0)

950 powerOf10 method

n 0

result 1

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

result 10 * powerOf10(2)

750 powerOf10 method

n 2

result 10 * powerOf10(1)

800 powerOf10 method

n 1

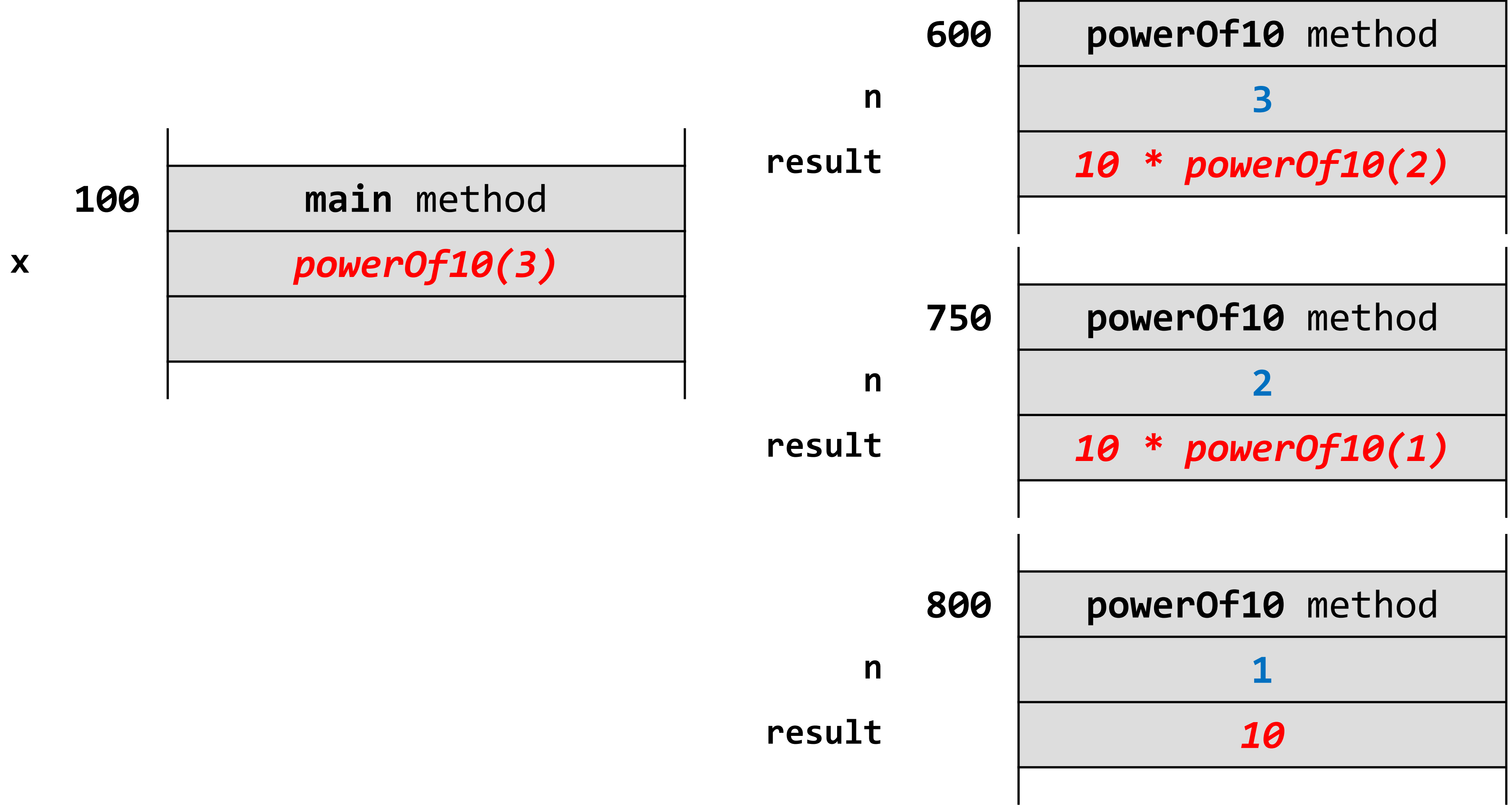
result 10 * 1

950 powerOf10 method

n 0

result 1

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

result 10 * powerOf10(2)

750 powerOf10 method

n 2

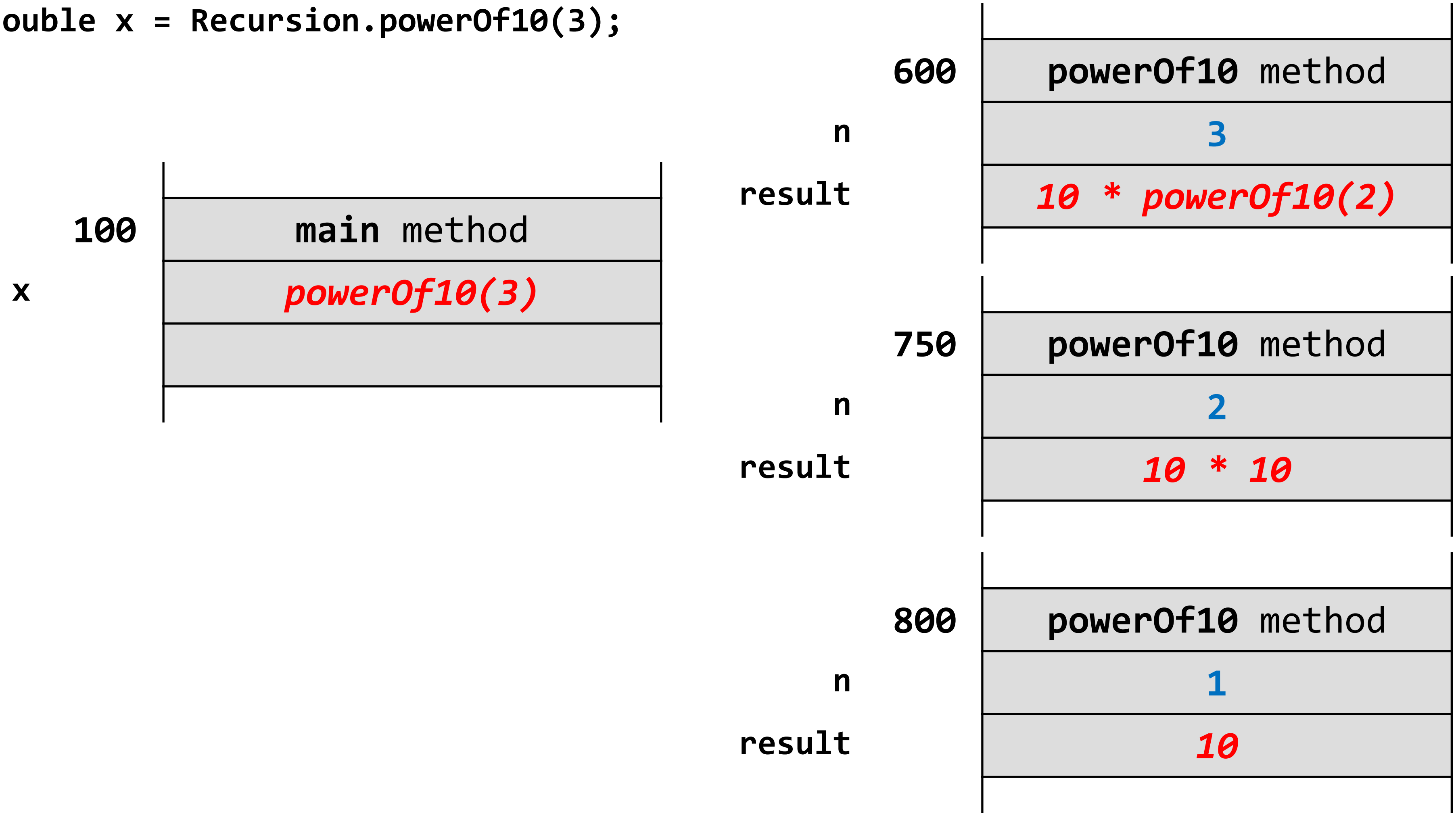
result 10 * powerOf10(1)

800 powerOf10 method

n 1

result 10

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



100 main method

x powerOf10(3)

600 powerOf10 method

n 3

result 10 * powerOf10(2)

750 powerOf10 method

n 2

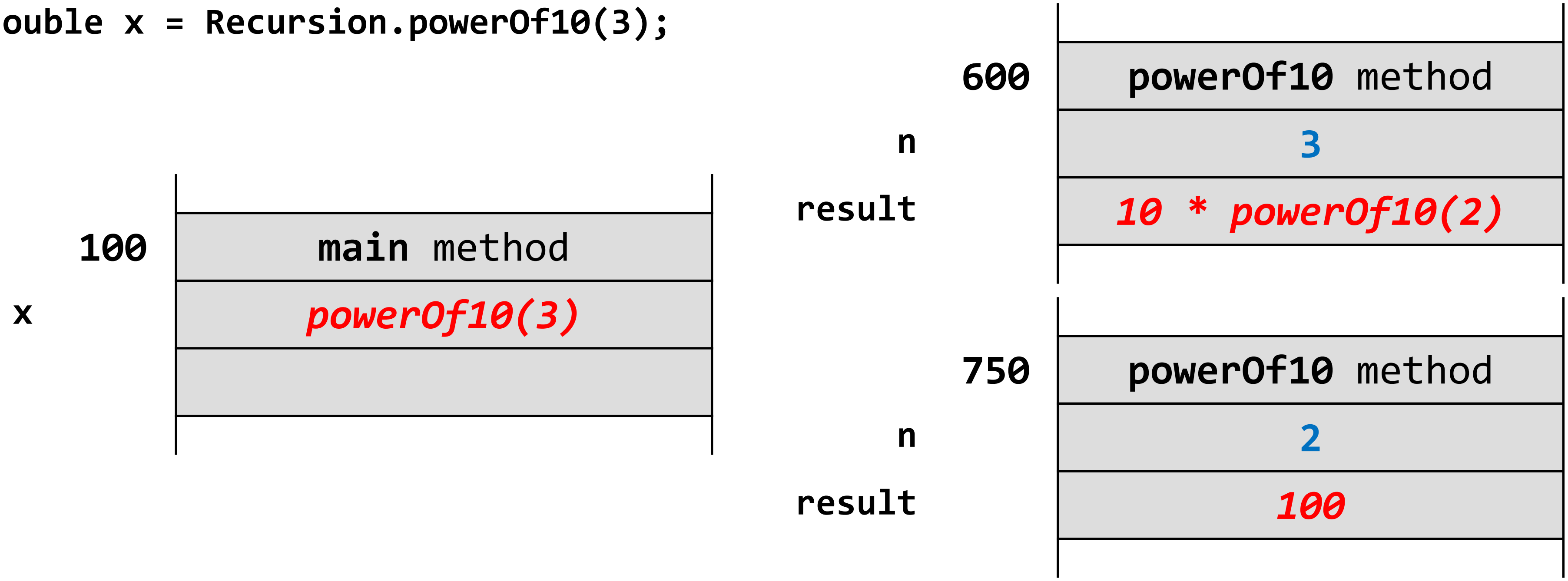
result 10 * 10

800 powerOf10 method

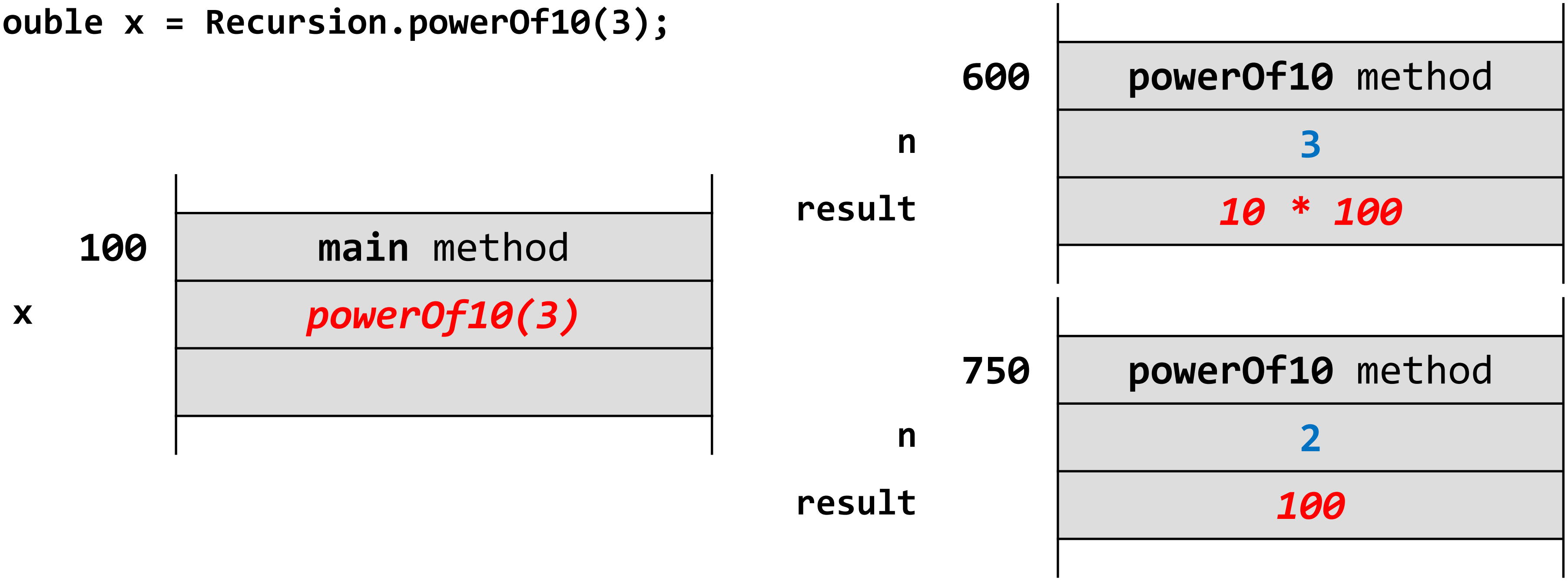
n 1

result 10

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



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



100 main method

x *powerOf10(3)*

600 powerOf10 method

n 3

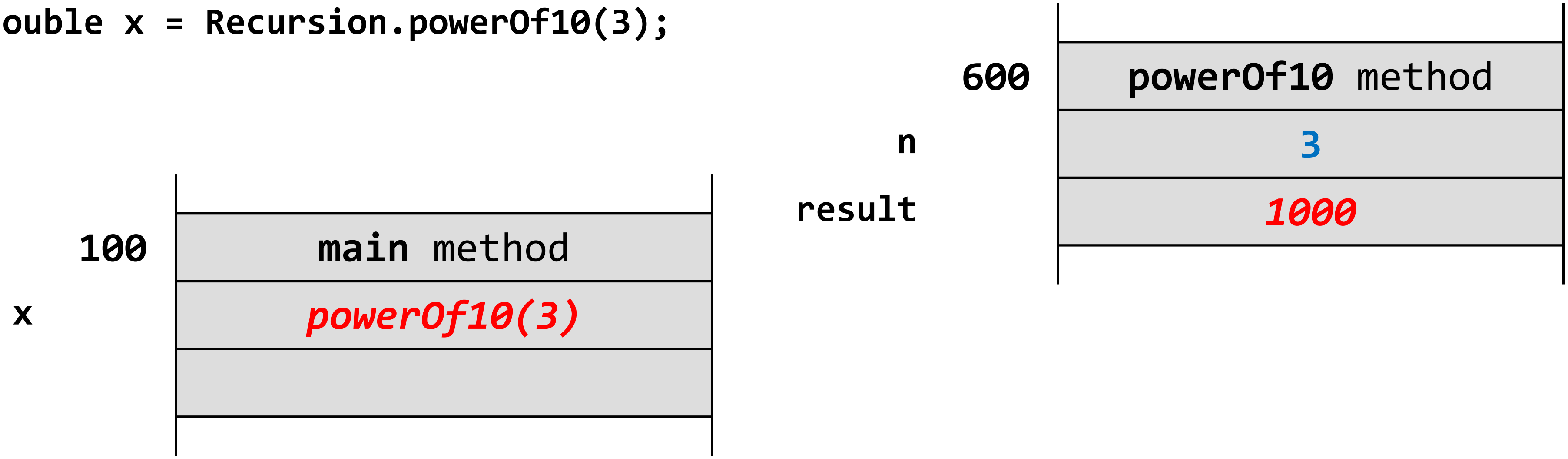
result *10 * 100*

750 powerOf10 method

n 2

result *100*

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



600 **powerOf10** method

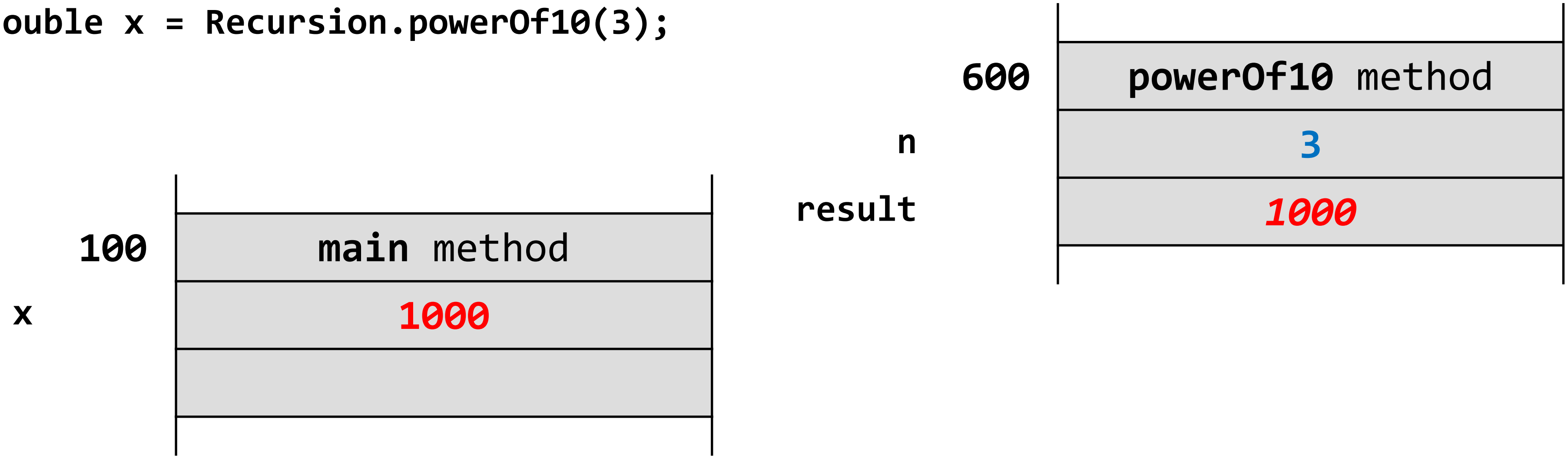
n 3

result *1000*

100 **main** method

x *powerOf10(3)*

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



600 powerOf10 method

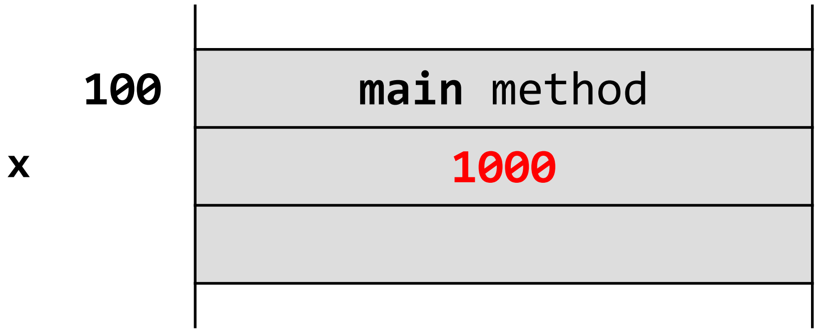
n 3

result 1000

100 main method

x 1000

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



100

main method

x

1000

– recursive methods that solve very large problems may exhaust call stack memory

⟹ StackOverflowException

Recursion and collections

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)
```

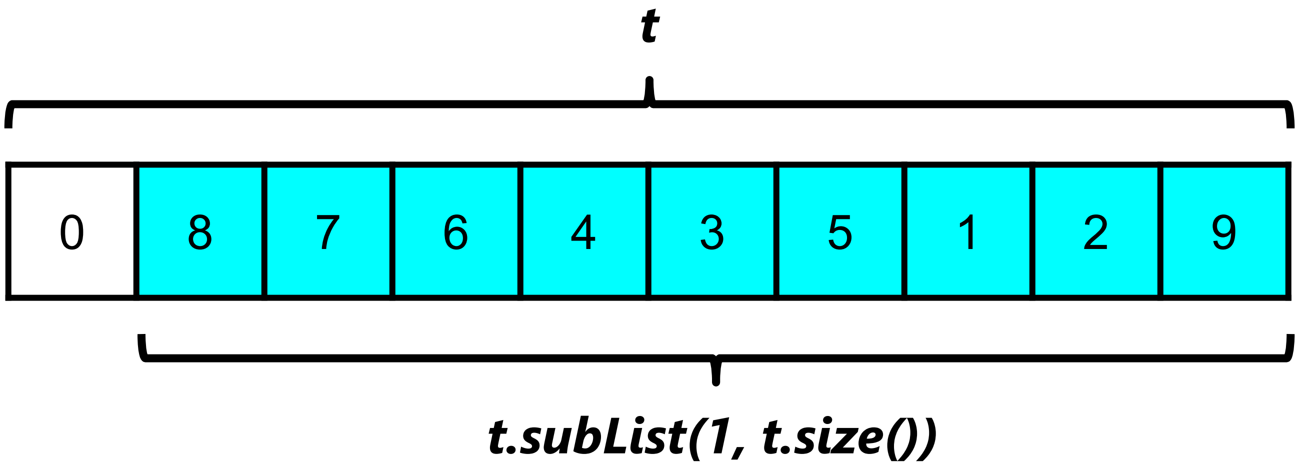
returns a List

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



t

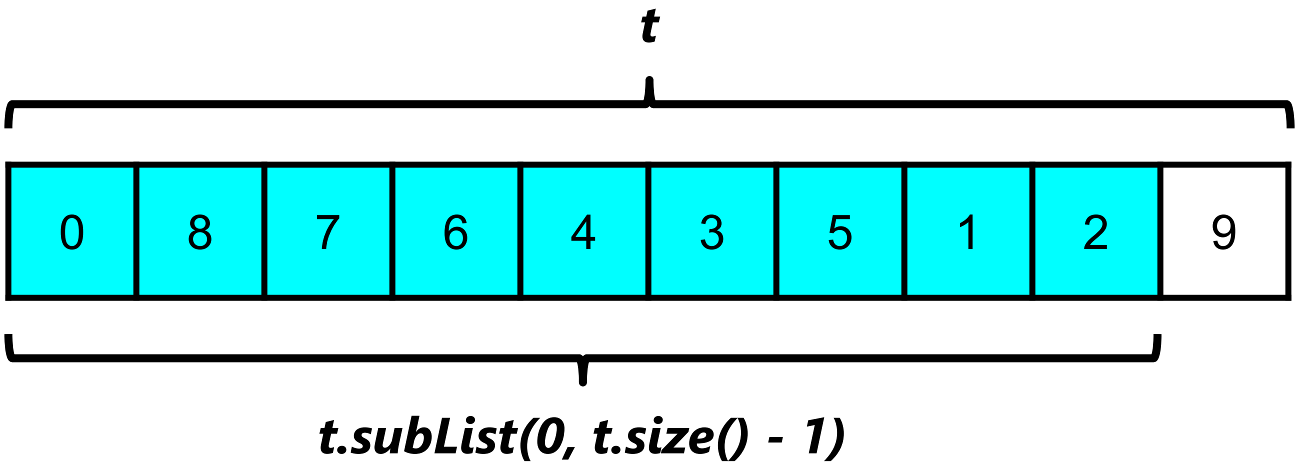
0	8	7	6	4	3	5	1	2	9

t.subList(1, t.size())

```
List<Integer> u = t.subList(1, t.size());
int first_u = u.get(0);                        // 8
int last_u = u.get(u.size() – 1);              // 9
```

`subList` examples

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



t

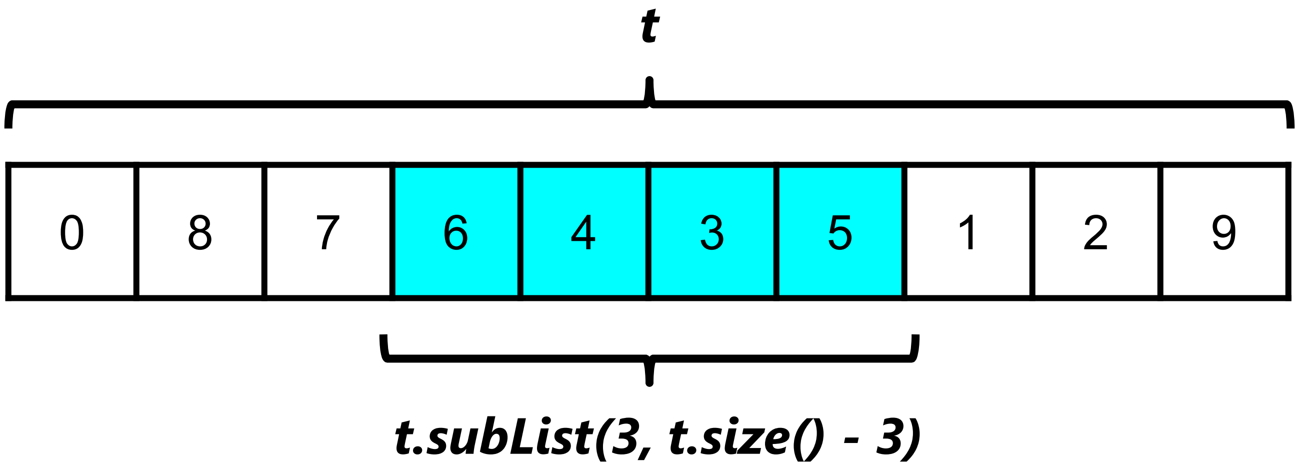
0	8	7	6	4	3	5	1	2	9

t.subList(0, t.size() - 1)

```
List<Integer> u = t.subList(0, t.size() - 1);
int first_u = u.get(0);                            // 0
int last_u = u.get(u.size() – 1);                  // 2
```

`subList` examples

- the sub-list excluding the first 3 and last 3 elements of the original list



t

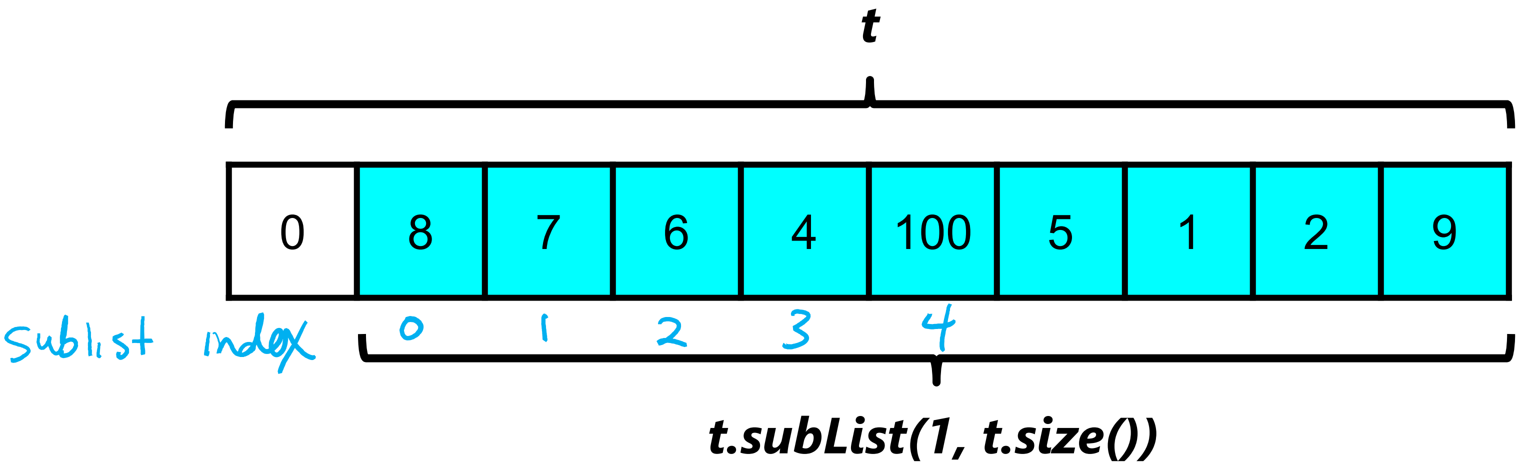
0	8	7	6	4	3	5	1	2	9

t.subList(3, t.size() - 3)

```
List<Integer> u = t.subList(3, t.size() - 3);
int first_u = u.get(0);                        // 6
int last_u = u.get(u.size() – 1);              // 5
```

subList examples

- modifying an element using the sublist modifies the element of the original list



t

0	8	7	6	4	100	5	1	2	9

sublist index 0 1 2 3 4

t.subList(1, t.size())

```
List<Integer> u = t.subList(1, t.size());
u.set(4, 100);                                   // set element at index 4 of u
int val_in_t = t.get(5);                         // 100
```

modifying the sublist modifies the original list

Recursively Search a List

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

size of the searched list decreases by 1

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

sublist

size of searched list decreases by 1

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

sublist

size of searched list decreases by 1

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

sublist

```
→ "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

```
public class Recursion {

  public static <T> boolean contains(T element, List<T> t) {
    boolean result;
    if (t.size() == 0) {                        // base case
      result = false;
    }
    else if (t.get(0).equals(element)) {   // base case
      result = true;
    }




  }
}
```

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

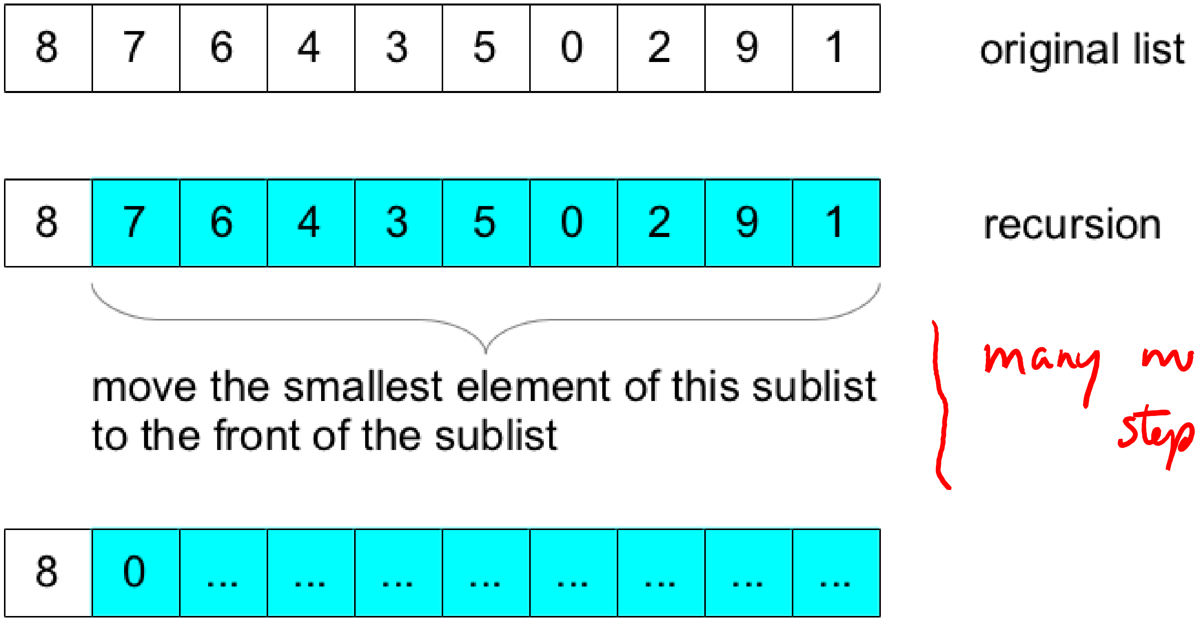
```
public class Recursion {

 public static <T> boolean contains(T element, List<T> t) {
  boolean result;
  if (t.size() == 0) {                        // base case
   result = false;
  }
  else if (t.get(0).equals(element)) {   // base case
   result = true;
  }
  else {                        // recursive call
   result = Recursion.contains(element, t.subList(1, t.size()));
  }
  return result;
 }
}
```

- search the rest of the list for element
- size of the sublist is t.size()-1

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

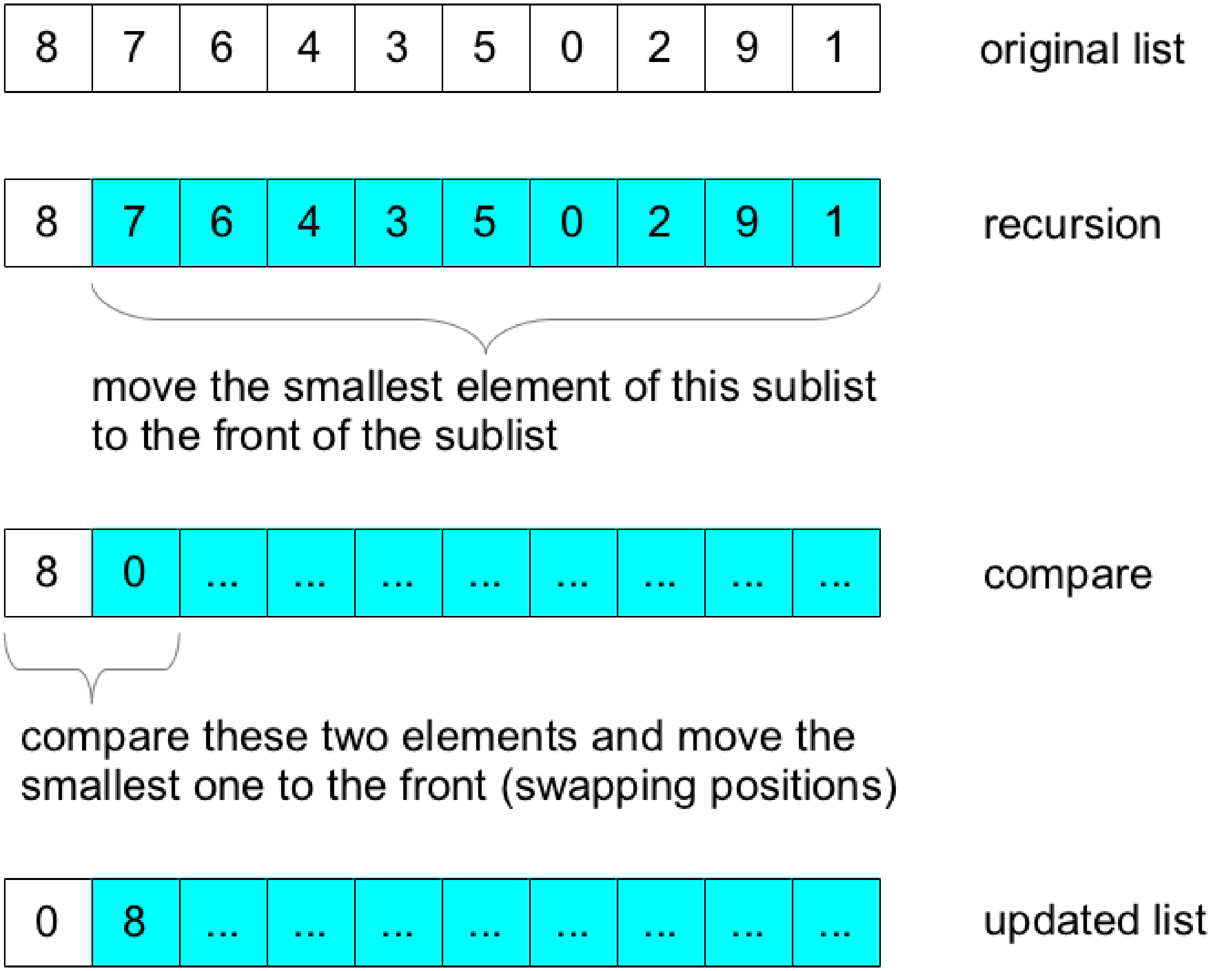


8	7	6	4	3	5	0	2	9	1	original list
8	7	6	4	3	5	0	2	9	1	recursion
8	0	...	...	...	...	...	...	...	...	

move the smallest element of this sublist to the front of the sublist

many missing steps

Recursively Move Smallest to Front



8	7	6	4	3	5	0	2	9	1	original list
8	7	6	4	3	5	0	2	9	1	recursion

move the smallest element of this sublist
to the front of the sublist

8	0	...	...	...	...	...	...	...	...	compare

compare these two elements and move the
smallest one to the front (swapping positions)

0	8	...	...	...	...	...	...	...	...	updated list

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 Recursion {

  public static void minToFront(List<Integer> t) {
  if (t.size() < 2) {
    return;
  }



  }
}
```

if the list is empty or has size == 1
there is nothing to do

* make sure your method handles the case where t is empty

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 Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Recursion.minToFront(t.subList(1, t.size()));








  }
}
```

moves smallest element in subList to front of subList

Recursively Move Smallest to Front

- compare and update?

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Recursion.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

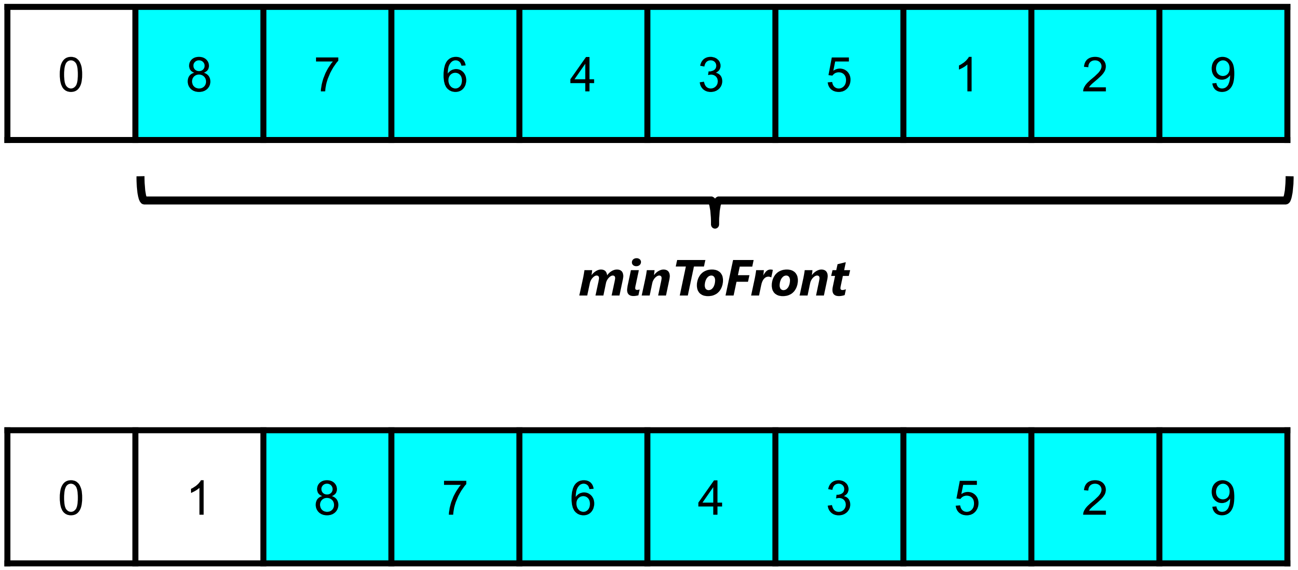
don't do this :

```
if (t.get(1) < t.get(0)) {
    t.set(0, t.get(1));
    t.set(1, t.get(0));
}
```

- Lab test 3 material stops here

Sorting the List

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

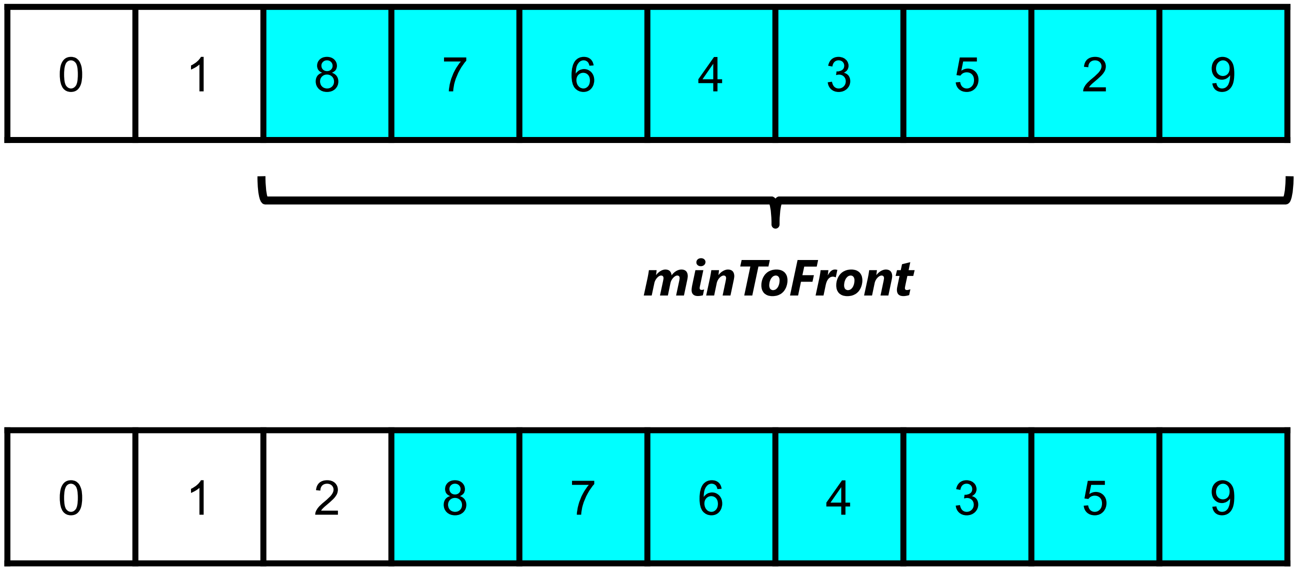


0	8	7	6	4	3	5	1	2	9

0	1	8	7	6	4	3	5	2	9

Sorting the List

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



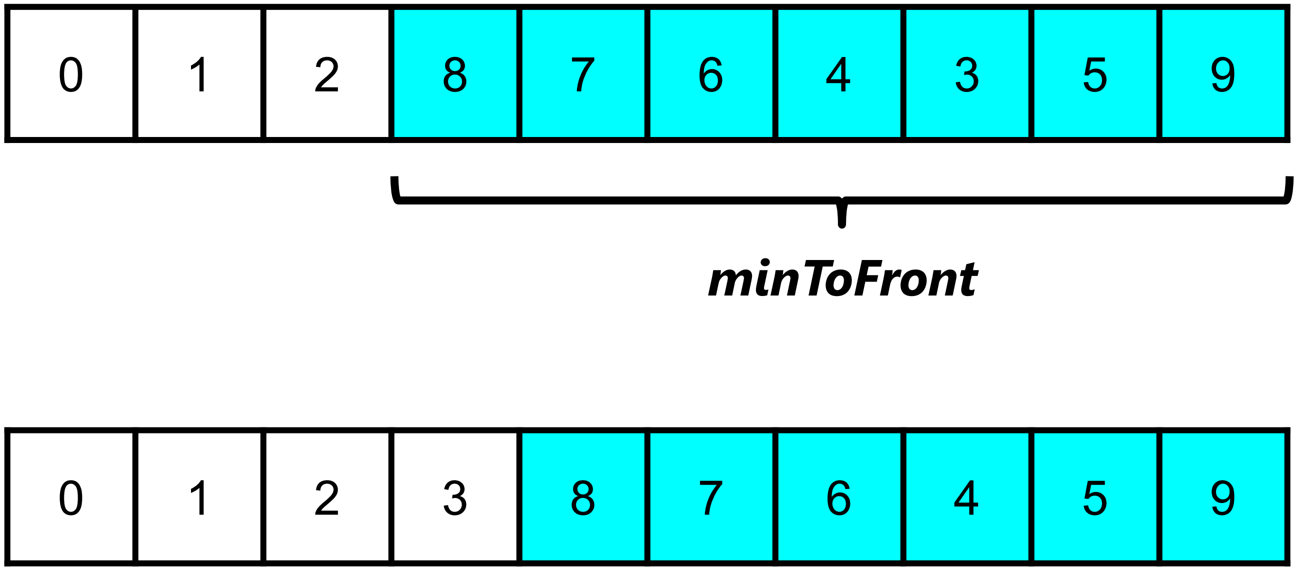
0	1	8	7	6	4	3	5	2	9

minToFront

0	1	2	8	7	6	4	3	5	9

Sorting the List

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



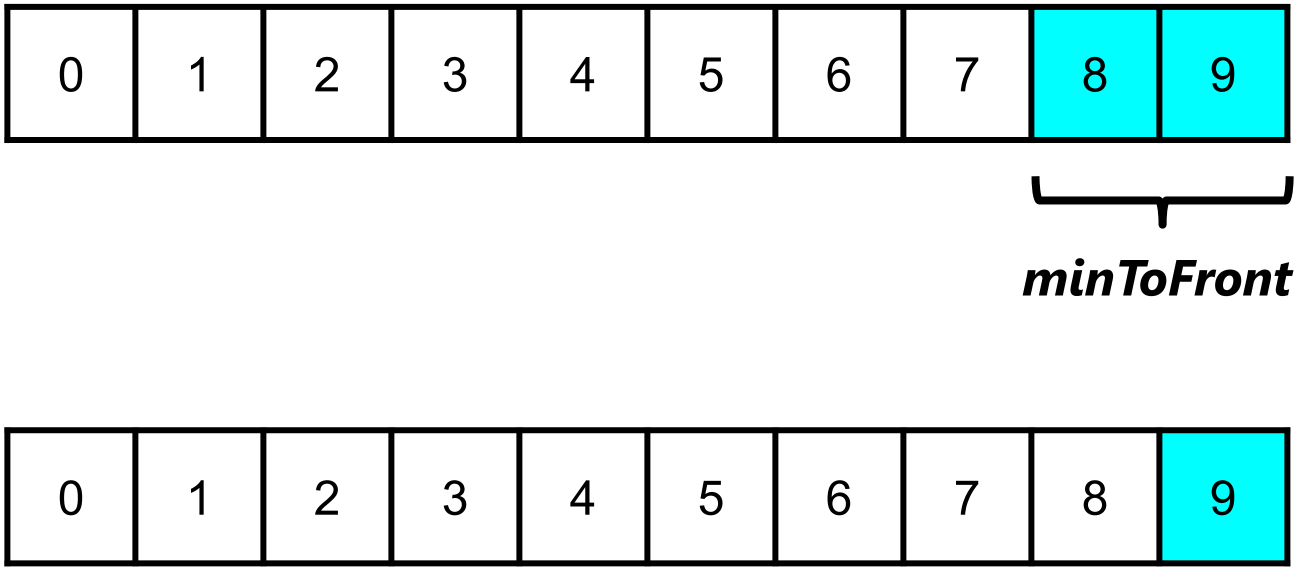
0	1	2	8	7	6	4	3	5	9

minToFront

0	1	2	3	8	7	6	4	5	9

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 Recursion {

  // minToFront not shown

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


}
```

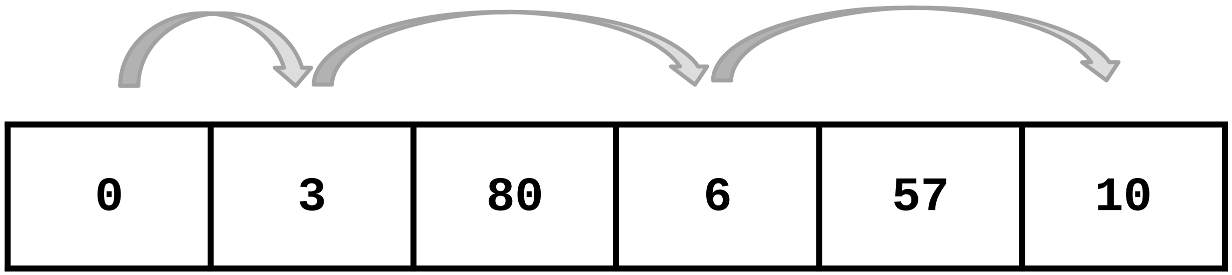
subList size is 1 smaller than t

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



0	3	80	6	57	10

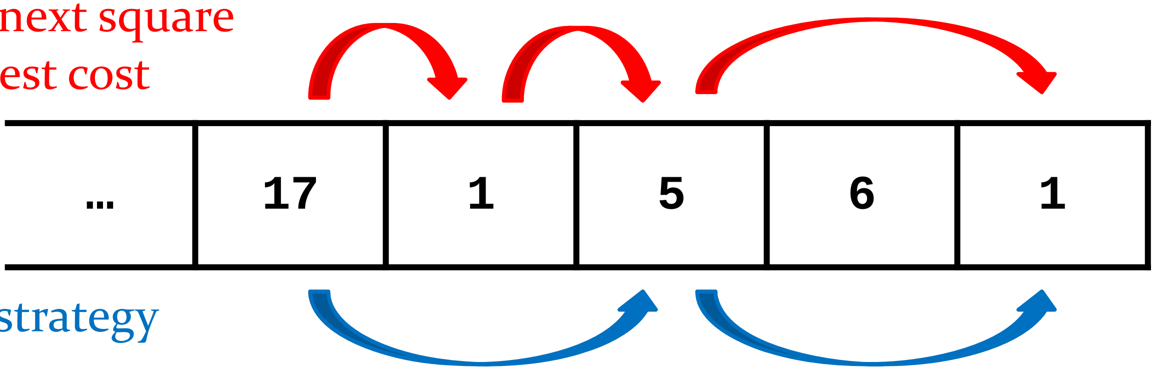
- solution for example:
  - move 1 square
  - move 2 squares
  - move 2 squares
    - □ total cost = 0 + 3 + 6 + 10 = 19

- 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

move to next square
with lowest cost



…	17	1	5	6	1

cost 17+1+5+1=24

optimal strategy

cost 17+5+1=23

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

```
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);
  }




}
```

base case #1 ; board size == 2

base case #2, board size == 3

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

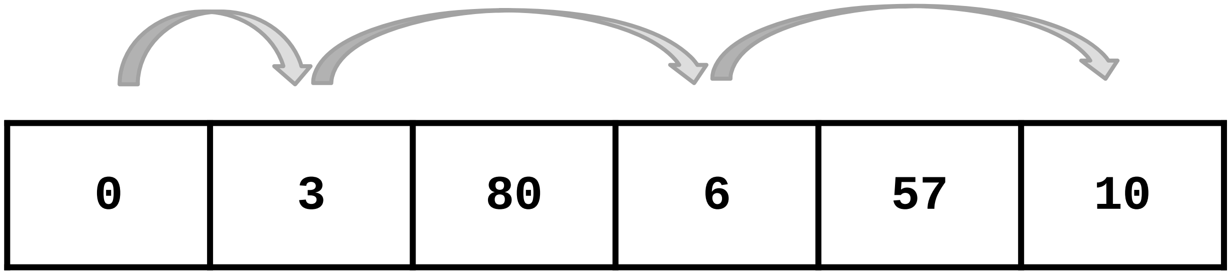
```
public static int cost(List<Integer> board) {
  if (board.size() == 2) {
    return board.get(0) + board.get(1);
  }
  if (board.size() == 3) {
    return board.get(0) + board.get(2);
  }
  List<Integer> afterOneStep = board.subList(1, board.size()); / board after moving 1 step
  List<Integer> afterTwoStep = board.subList(2, board.size()); - board after moving 2 steps
  int c = board.get(0); - cost of current square
  return c + Math.min(cost(afterOneStep), cost(afterTwoStep));
}
```

cost after moving 1 step

cost after moving 2 steps

Jump It

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



0	3	80	6	57	10

the method produces the list `[1, 2, 2]`

- consider using the following modified signature

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

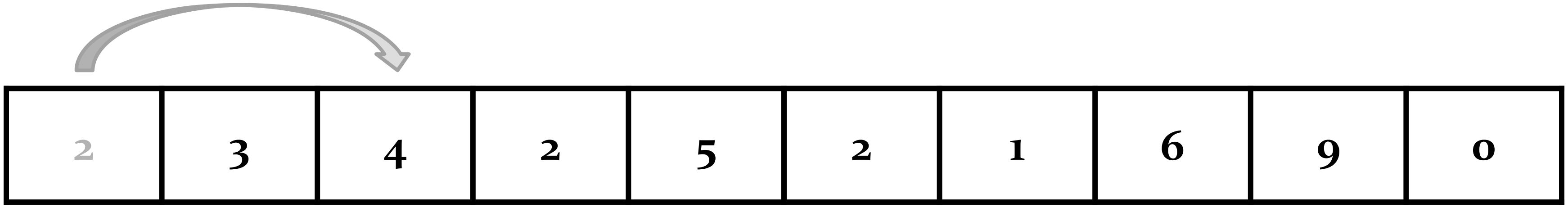
- the Jump It problem has a couple of nice properties:
  - the rules of the game make it impossible to move to the same square twice
  - the rules of the games make it impossible to try to move off of the board
- consider the following problem

- given a list of non-negative integer values:

2	3	4	2	5	2	1	6	9	0

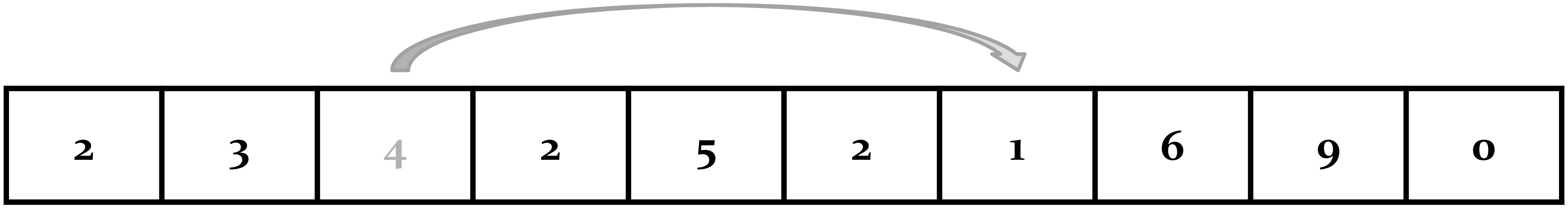
  - starting from the first element try to reach the last element (whose value is always zero)
  - you may move left or right by the number of elements equal to the value of the element that you are currently on
  - you may not move outside the bounds of the list

Solution 1



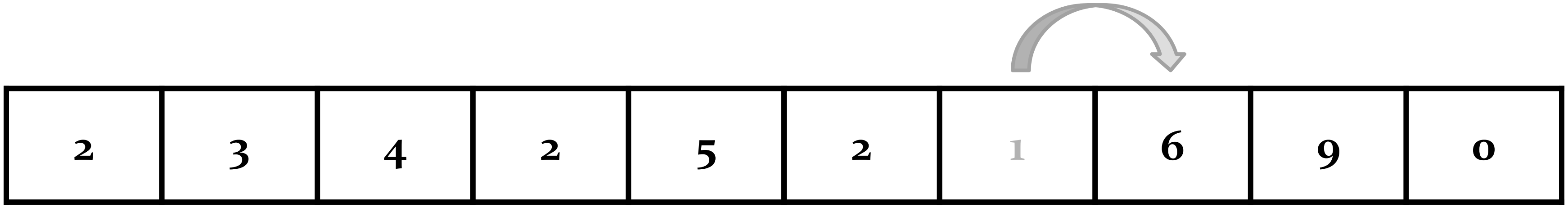
2	3	4	2	5	2	1	6	9	0

Solution 1



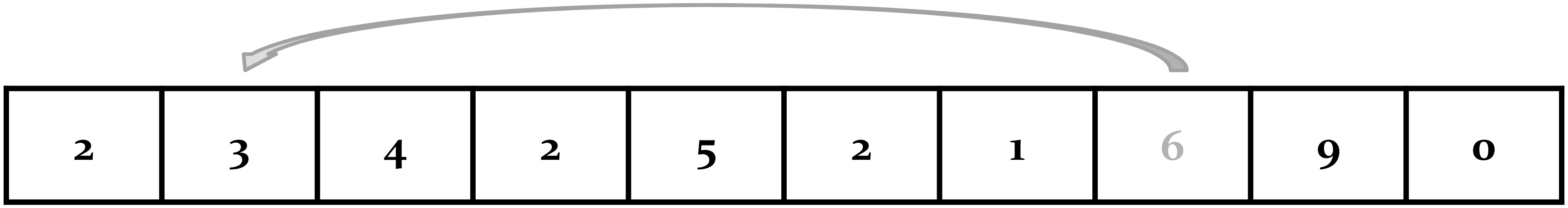
2	3	4	2	5	2	1	6	9	0

Solution 1



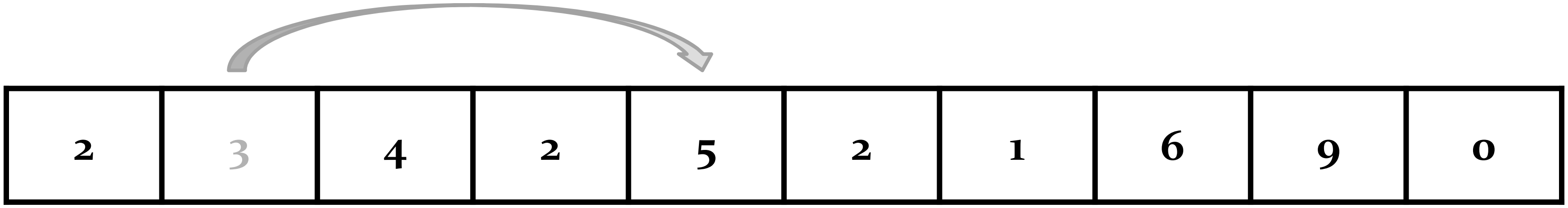
2	3	4	2	5	2	1	6	9	0

Solution 1



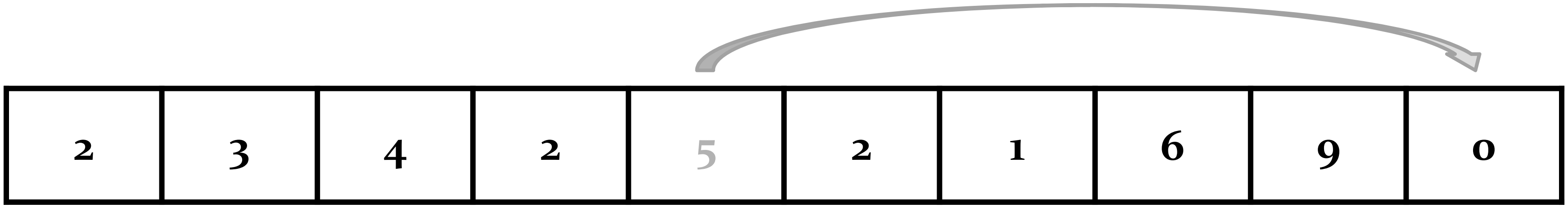
2	3	4	2	5	2	1	6	9	0

Solution 1



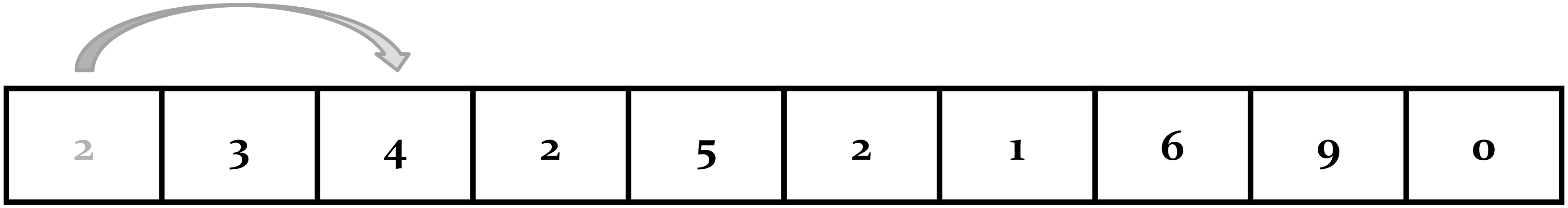
2	3	4	2	5	2	1	6	9	0

Solution 1



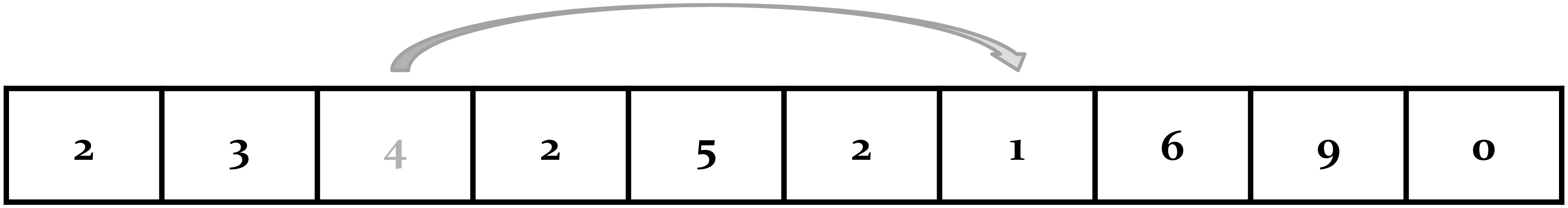
2	3	4	2	5	2	1	6	9	0

Solution 2



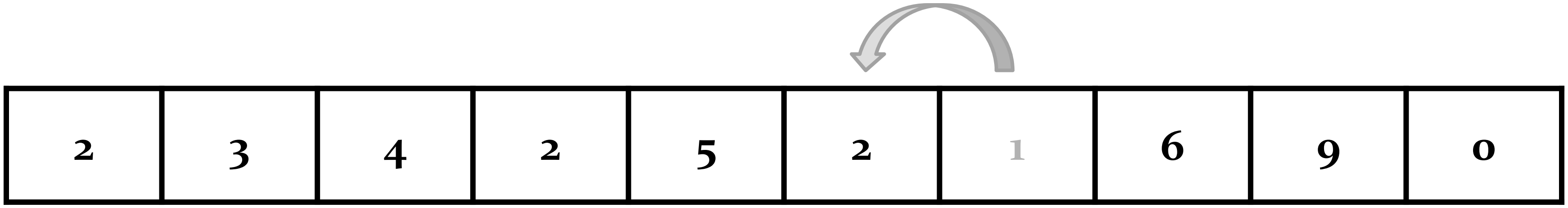
2	3	4	2	5	2	1	6	9	0

Solution 2



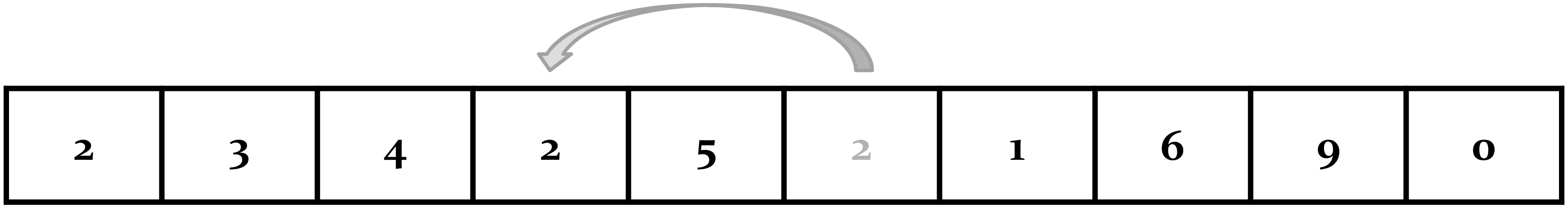
2	3	4	2	5	2	1	6	9	0

Solution 2



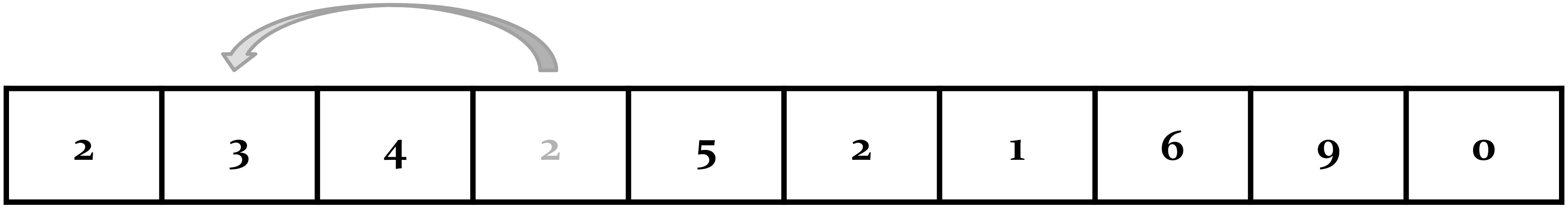
2	3	4	2	5	2	1	6	9	0

Solution 2



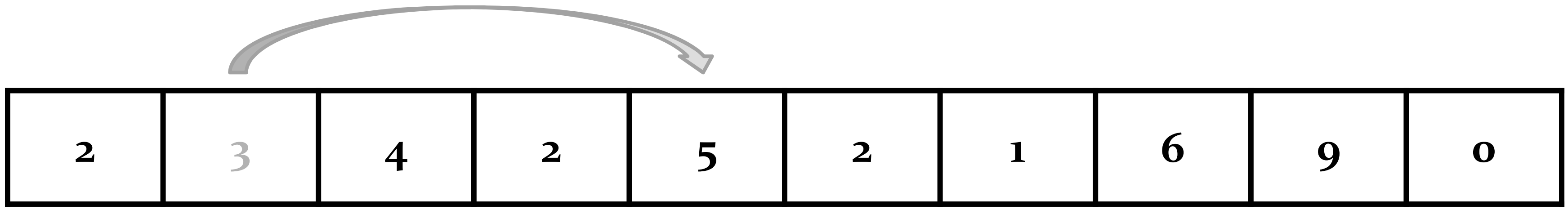
2	3	4	2	5	2	1	6	9	0

Solution 2



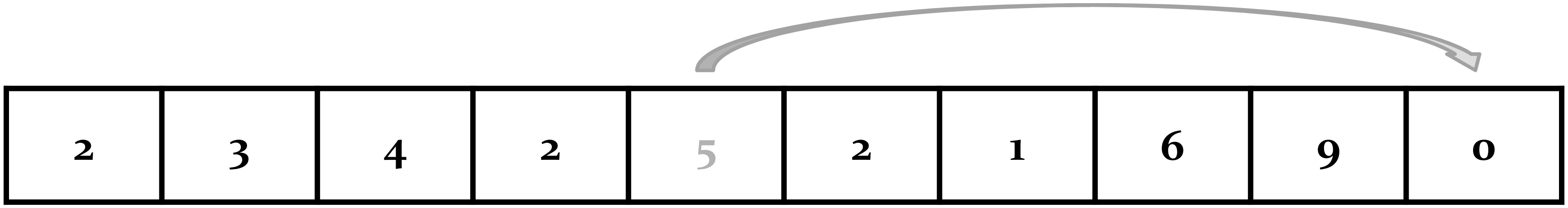
2	3	4	2	5	2	1	6	9	0

Solution 2



2	3	4	2	5	2	1	6	9	0

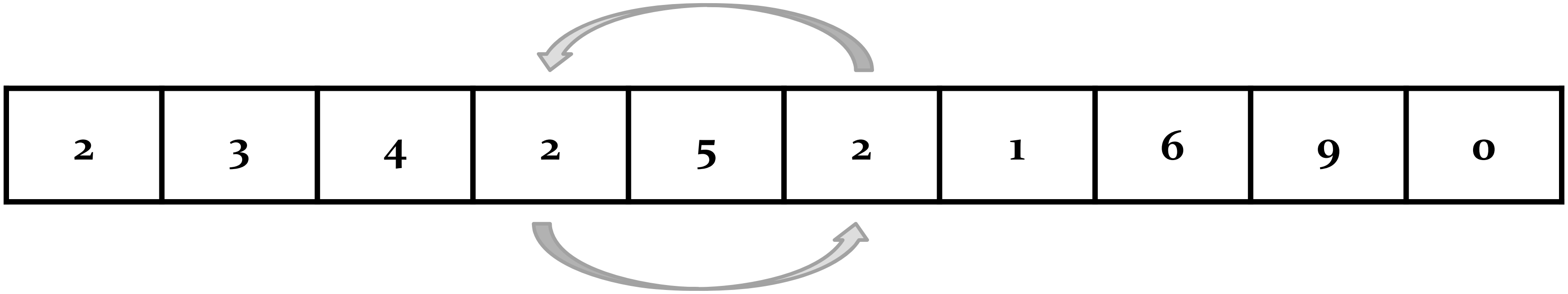
Solution 2



2	3	4	2	5	2	1	6	9	0

Cycles

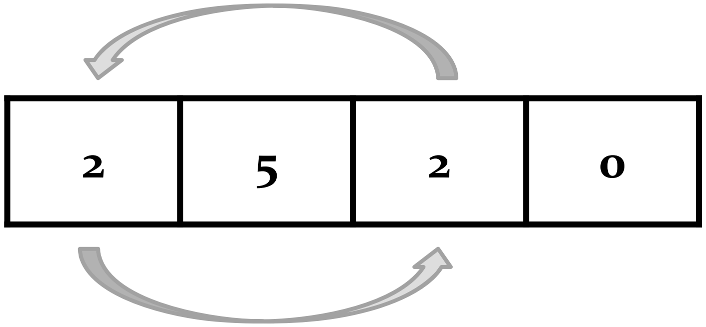
- it is possible to find cycles where a move takes you back to a square that you have already visited



2	3	4	2	5	2	1	6	9	0

Cycles

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



2	5	2	0

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? — solution allows you to step on a square only once
    - how do we indicate this?

mark the square on the board with a special value

i.e. −1

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;










}
```

— # of steps we must move

– make a copy of the board

— indicate that the solution should not return to this square

```
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) {

    }



}
```

index of the square "value" steps to the left

```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);
    }







}
```

```
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);





}
```

```
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()) {

    }
}
```

index of square "value" steps to the right

```
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);
    }

}
```

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

    copy = new ArrayList<Integer>(board); — not necessary
    copy.set(index, -1); — not necessary
    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) {  — we've seen this square before
      return false;
    }
    if (index == board.size() - 1) {  — we're on the last square of
      return true;                              the board
    }
    // recursive cases go here...








}
```

Recursion: Computational Complexity

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Recursion.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

size of problem, n, is the number of elements in the list **t**

Estimating complexity

- the basic strategy for estimating complexity:
  1. for each line of code, estimate its number of elementary instructions
  2. for each line of code, determine how often it is executed
  3. determine the total number of elementary instructions

Elementary instructions

- what is an elementary instruction?
  - for our purposes, any expression that can be computed in a constant amount of time
- examples:
  - declaring a variable
  - assignment (=)
  - arithmetic (+, -, *, /, %)
  - comparison (<, >, ==, !=)
  - Boolean expressions (||, &&, !)
  - if, else
  - return statement

Estimating complexity

- count the number of elementary operations in each line of **`minToFront`**
  - assume that the following are all elementary operations:
    - **`t.size()`**
    - **`t.get(0)`**
    - **`t.get(1)`**
    - **`t.set(0, ...)`**
    - **`t.set(1, ...)`**
    - **`t.subList(x, y)`**
  - leave the line with the recursive call blank for now

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Recursion.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

number of elementary ops

3
1
???
3
3
2
1
1

Estimating complexity

- for each line of code, determine how often it is executed

Recursively Move Smallest to Front

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {                                    1
      return;                                              1 or 0
    }
    Recursion.minToFront(t.subList(1, t.size()));          1 or 0
    int first = t.get(0);                                  1 or 0
    int second = t.get(1);                                 1 or 0
    if (second < first) {                                  1 or 0
      t.set(0, second);                                    1 or 0
      t.set(1, first);                                     1 or 0
    }
  }
}
```

Total number of operations

- before we can determine the total number of elementary operations, we need to count the number of elementary operations arising from the recursive call
- let $T(n)$ be the total number of elementary operations required by **minToFront(t)** for a list t with size n

Total number of operations

```
public class Recursion {

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



    Recursion.minToFront(t.subList(1, t.size()));
                                      └────┬────┘
                             1 elementary operation




  }
}
```

Total number of operations

```
public class Recursion {

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



    Recursion.minToFront(t.subList(1, t.size()));
```

1 elementary operation

```
  }
}
```

Total number of operations

```
public class Recursion {

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



    Recursion.minToFront(t.subList(1, t.size()));



  }
}
```

$T(n-1)$ elementary operations

because the sublist has size $(n-1)$

Total number of operations

```
public class Recursion {

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



    Recursion.minToFront(t.subList(1, t.size()));
```

$$\left.\begin{array}{r} T(n-1)\text{ elementary operations}\\ 1\text{ elementary operation}\\ 1\text{ elementary operation}\end{array}\right\} = T(n-1)+2$$

```
  }
}
```

Total number of operations

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
   if (t.size() < 2) {
     return;
   }
   Recursion.minToFront(t.subList(1, t.size()));
   int first = t.get(0);
   int second = t.get(1);
   if (second < first) {
     t.set(0, second);
     t.set(1, first);
   }
  }
}
```

these lines run if the base case is true

Total number of operations

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {                                  3 * 1
      return;                                            1 * 1
    }
    Recursion.minToFront(t.subList(1, t.size()));     (T(n-1) + 2) * 1
    int first = t.get(0);                                3 * 1
    int second = t.get(1);                               3 * 1
    if (second < first) {                                2 * 1
      t.set(0, second);                                  1 * 1
      t.set(1, first);                                   1 * 1
    }
  }
}
```

Total number of operations

- base cases
  - $T(0) = T(1) = 4$

the # of elementary operation for a list of size 0 or a list of size 1

Total number of operations

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
```

this line runs if the base case is not true

```
      return;
    }
    Recursion.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
```

these lines run if the base case is not true

```
      t.set(0, second);
      t.set(1, first);
```

these lines might run if the base case is not true

```
    }
  }
}
```

Total number of operations

- when counting the total number of operations, we often consider the worst case scenario
  - let’s assume that the lines that might run always run

Total number of operations

```
public class Recursion {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {                                    3 * 1
      return;                                              1 * 1
    }
    Recursion.minToFront(t.subList(1, t.size()));     (T(n-1) + 2) * 1
    int first = t.get(0);                                  3 * 1
    int second = t.get(1);                                 3 * 1
    if (second < first) {                                  2 * 1
      t.set(0, second);                                    1 * 1
      t.set(1, first);                                     1 * 1
    }
  }
}
```

Total number of operations

- base cases
  - $T(0) = T(1) = 4$
- recursive case
  - $T(n) = T(n-1) + 15$ 16?

- the two equations above are called the *recurrence relation* for **minToFront**

Selection Sort

```
public class Recursion {

  // minToFront not shown

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

}
```

number of elementary ops?

3

?

$T(n-1) + 2$

let $T(n)$ be the number of elementary operations for selectionSort(t) for a list t of size n

Total number of operations

- base cases
  - $T(0) = T(1) = 4$
- recursive case
  - $T(n) = T(n-1) + 15$

- the two equations above are called the *recurrence relation* for **minToFront**
- let's try to solve the recurrence relation

Solving the recurrence relation

$$T(0) = 4$$
$$T(1) = 4$$
$$T(n) = T(n-1) + 15$$

- if we knew $T(n-1)$ we could solve for $T(n)$

$$\begin{aligned} T(n) &= T(n-1) + 15 \\ &= (T(n-2) + 15) + 15 \\ &= T(n-2) + 2(15) \end{aligned}$$

$$T(n-1) = T(n-2) + 15$$

Solving the recurrence relation

$$T(0) = 4$$
$$T(1) = 4$$
$$T(n) = T(n-1) + 15$$

- if we knew $T(n-2)$ we could solve for $T(n)$

$$\begin{aligned} T(n) &= T(n-1) + 15 \\ &= (T(n-2) + 15) + 15 \\ &= T(n-2) + 2(15) \\ &= (T(n-3) + 15) + 2(15) \\ &= T(n-3) + 3(15) \end{aligned}$$

$$T(n-1) = T(n-2) + 15$$

$$T(n-2) = T(n-3) + 15$$

Solving the recurrence relation

$T(0) = 4$
$T(1) = 4$
$T(n) = T(n-1) + 15$

- if we knew $T(n-3)$ we could solve for $T(n)$

$$
\begin{aligned}
T(n) &= T(n-1) + 15 \\
&= (T(n-2) + 15) + 15 \\
&= T(n-2) + 2(15) \\
&= (T(n-3) + 15) + 2(15) \\
&= T(n-3) + 3(15) \\
&= (T(n-4) + 15) + 3(15) \\
&= T(n-4) + 4(15)
\end{aligned}
$$

$T(n-1) = T(n-2) + 15$

$T(n-2) = T(n-3) + 15$

$T(n-3) = T(n-4) + 15$

Solving the recurrence relation

$$T(0) = 4$$
$$T(1) = 4$$
$$T(n) = T(n-1) + 15$$

- there is clearly a pattern

$$T(n) = T(n-k) + k(15)$$

Solving the recurrence relation

$$T(0) = 4$$
$$T(1) = 4$$
$$T(n) = T(n-1) + 15$$

- substitute $\boxed{k = n - 1}$ so that we reach a base case

$$\begin{aligned} T(n) &= T(n-k) + k(15) \\ &= T\big(n-(n-1)\big) + (n-1)(15) \\ &= T(1) + 15n - 15 \\ &= 4 + 15n - 15 \\ &= 15n - 11 \in \underline{O(n)} \end{aligned}$$

Big-O notation

- Proof: $f(n) = 15n - 11, g(n) = n$

For $n \geq 1$, $f(n) > 0$ and $g(n) \geq 0$; therefore, we do not need to consider the absolute values. We need to find M and m such that the following is true:

$$\underline{15n - 11 < Mn \text{ for all } n > m}$$

For $n > 0$ we have:

$$\frac{15n-11}{n} < \frac{15n}{n} = 15$$

$$\frac{15n-11}{n} < \frac{15n+11}{n} < \frac{15n+11n}{n} = \frac{26n}{n} = \underset{M}{\boxed{26}}$$

$\therefore$ $\underline{15}n - 11 < \underbrace{15}_{M}n$ for all $n > \underbrace{0}_{m}$ and $T(n) \in O(n)$

Try to solve the recurrence relation

$T(1) = 1$
$T(n) = T(n-1) + 3n$

try this and see what happens

~ leads to having to compute some sums of series

$T(n) \in O(n^2)$

Try to solve the recurrence relation

$T(1) = 7$

$T(n) = T\left(\frac{n}{2}\right) + 1$

start with:

$T\left(\frac{n}{2}\right) = T\left(\frac{\frac{n}{2}}{2}\right) + 1$

$= T\left(\frac{n}{4}\right) + 1$

$\therefore T(n) = \left(T\left(\frac{n}{4}\right) + 1\right) + 1$

$= T\left(\frac{n}{4}\right) + 2$

"the number of elementary operations needed to solve a problem of size n is equal to the number of operations needed to solve a problem of size $\frac{n}{2}$ plus 1 more operation"

$T(n) \in O(\log_2 n)$

Try to solve the recurrence relation

$T(0) = 3$
$T(1) = 3$
$T(n) = T(n-1) + Mn + 5$

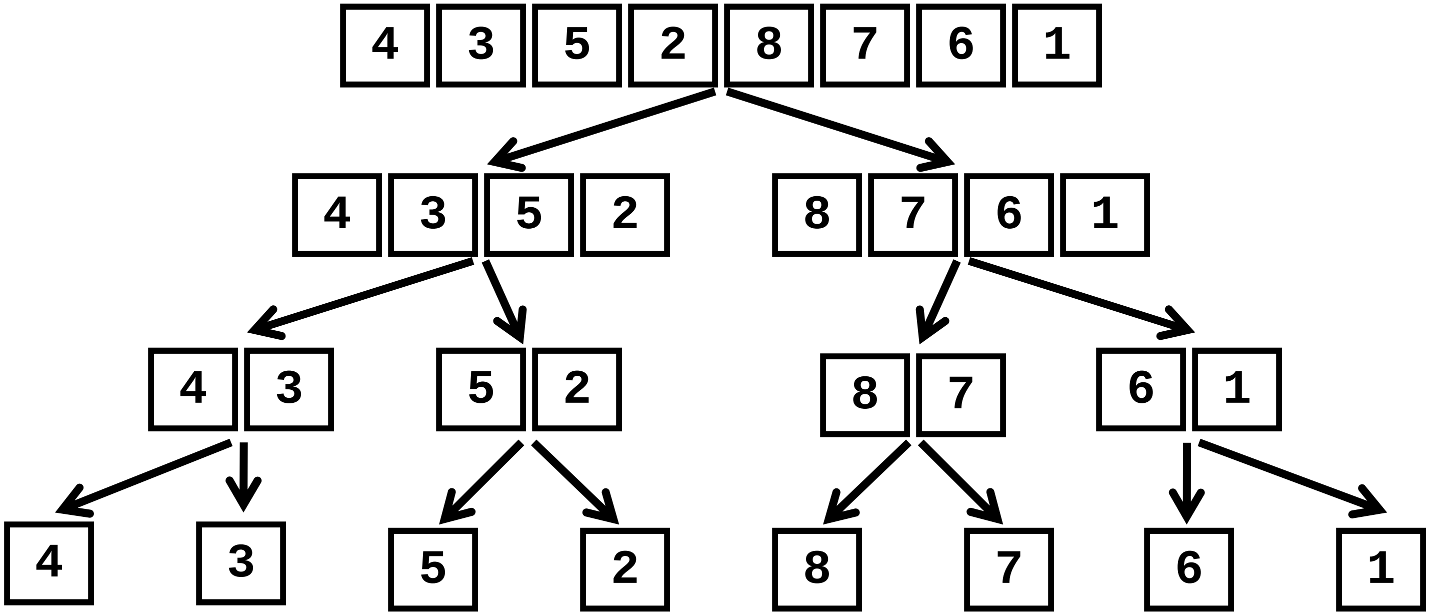
- again leads to computing some sums of series

Divide and Conquer

- divide and conquer algorithms typically recursively divide a problem into several smaller sub-problems until the sub-problems are small enough that they can be solved directly

Merge Sort

- merge sort is a divide and conquer algorithm that sorts a list of numbers by recursively splitting the list into two halves



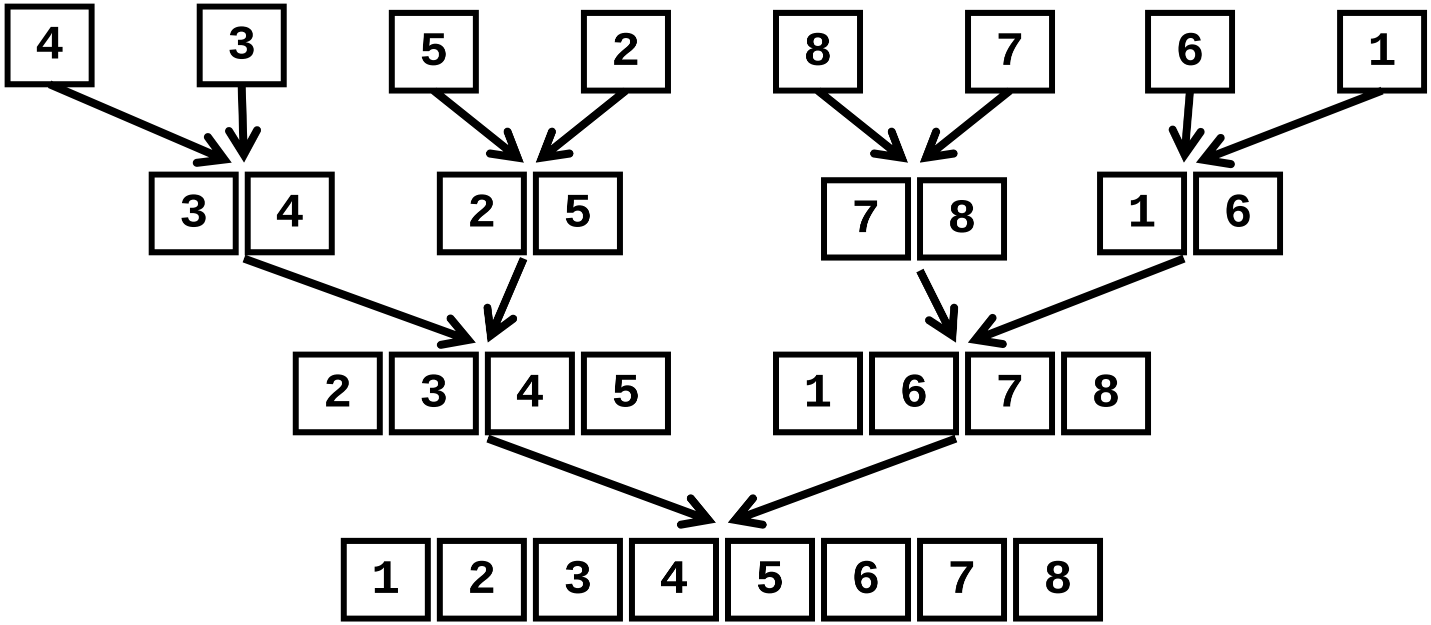
4 3 5 2 8 7 6 1

4 3 5 2 8 7 6 1

4 3 5 2 8 7 6 1

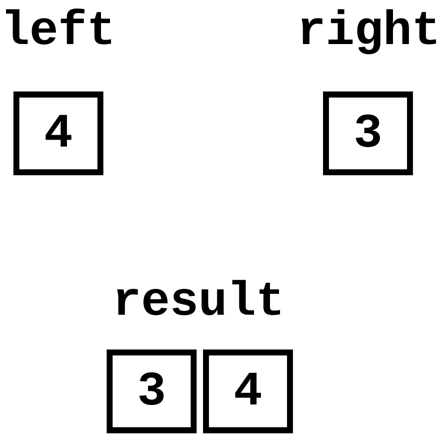
4 3 5 2 8 7 6 1

- the split lists are then merged into sorted sub-lists



Merging Sorted Sub-lists

- two sub-lists of length 1



left

4

right

3

result

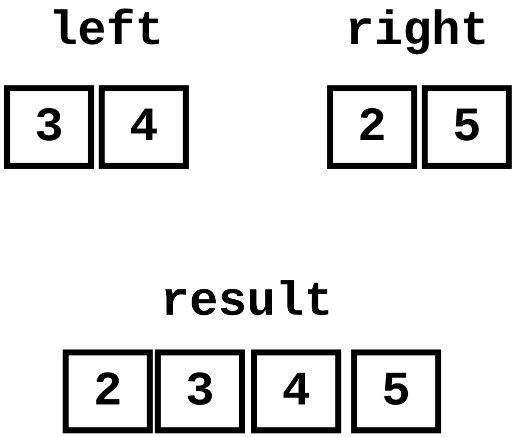
3 4

1 comparison
2 copies

```
LinkedList<Integer> result = new LinkedList<Integer>();

int fL = left.getFirst();
int fR = right.getFirst();
if (fL < fR) {
  result.add(fL);
  left.removeFirst();
}
else {
  result.add(fR);
  right.removeFirst();
}
if (left.isEmpty()) {
  result.addAll(right);
}
else {
  result.addAll(left);
}
```

Merging Sorted Sub-lists

- two sub-lists of length 2



left

3 4

right

2 5

result

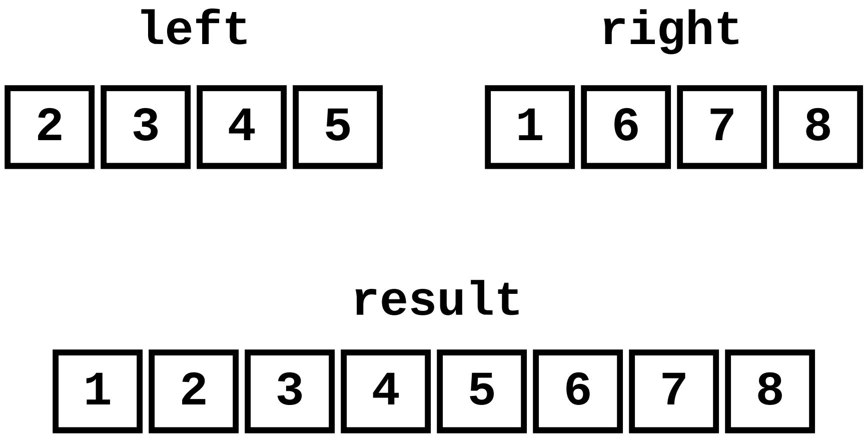
2 3 4 5

3 comparisons
4 copies

```
LinkedList<Integer> result = new LinkedList<Integer>();

while (left.size() > 0 && right.size() > 0 ) {
  int fL = left.getFirst();
  int fR = right.getFirst();
  if (fL < fR) {
    result.add(fL);
    left.removeFirst();
  }
  else {
    result.add(fR);
    right.removeFirst();
  }
}
if (left.isEmpty()) {
  result.addAll(right);
}
else {
  result.addAll(left);
}
```

Merging Sorted Sub-lists

- two sub-lists of length 4



left

2	3	4	5

right

1	6	7	8

result

1	2	3	4	5	6	7	8

5 comparisons
8 copies

Simplified Complexity Analysis

- in the worst case merging a total of **n** elements requires

  n – 1 comparisons **+**

  n copies

  = 2n – 1 total operations

- the worst-case complexity of merging is the order of *O(n)*

Informal Analysis of Merge Sort

- suppose the running time (the number of operations) of merge sort is a function of the number of elements to sort
  - let the function be $T(n)$
- merge sort works by splitting the list into two sub-lists (each about half the size of the original list) and sorting the sub-lists
  - this takes $\mathbf{2}T(n/\mathbf{2})$ running time
- then the sub-lists are merged
  - this takes $O(n)$ running time
- total running time $T(n) = \underbrace{\mathbf{2}T(n/\mathbf{2})}_{\text{sort 2 sublists of size } \left(\frac{n}{2}\right)} + \underbrace{O(n)}_{\text{merge}}$

Solving the Recurrence Relation

$$\begin{aligned}
T(n) \quad &\rightarrow \quad 2T(n/2) + O(n) \\
&\approx \quad 2T(n/2) + n \\
&= \quad 2[\ 2T(n/4) + n/2\ ] + n \\
&= \quad 4T(n/4) + 2n \\
&= \quad 4[\ 2T(n/8) + n/4\ ] + 2n \\
&= \quad 8T(n/8) + 3n \\
&= \quad 8[\ 2T(n/16) + n/8\ ] + 3n \\
&= \quad 16T(n/16) + 4n \\
&= \quad 2^k T(n/2^k) + kn
\end{aligned}$$

$T(n)$ approaches…

or $2T(\frac{n}{2}) + Mn$

Solving the Recurrence Relation

$$T(n) \quad = \qquad 2^k T(\underline{n/2^k}) + kn$$

- for a list of length $\mathbf{1}$ we know $T(\mathbf{1}) = \mathbf{1}$
  - if we can substitute $T(1)$ into the right-hand side of $T(n)$ we might be able to solve the recurrence
  - we have $T(n/\mathbf{2}^k)$ on the right-hand side, so we need to find some value of k such that

$$\underline{n/\mathbf{2}^k} = \mathbf{1} \;\Rightarrow\; \mathbf{2}^k = n \Rightarrow k = \log_2(n)$$

Solving the Recurrence Relation

$$= 2^k T(n/2^k) + kn$$

$$\begin{aligned} T(n) &= 2^{\log_2 n} T\left(n/2^{\log_2 n}\right) + n \log_2 n \\ &= n\, T(1) + n \log_2 n \\ &= n + n \log_2 n \\ &\in O(n \log_2 n) \end{aligned}$$

Quicksort

- quicksort, like mergesort, is a divide and conquer algorithm for sorting a list or array
- it can be described recursively as follows:
  1. choose an element, called the *pivot*, from the list
  2. reorder the list so that:
     - values less than the pivot are located before the pivot
     - values greater than the pivot are located after the pivot
  3. quicksort the sublist of elements before the pivot
  4. quicksort the sublist of elements after the pivot

Quicksort

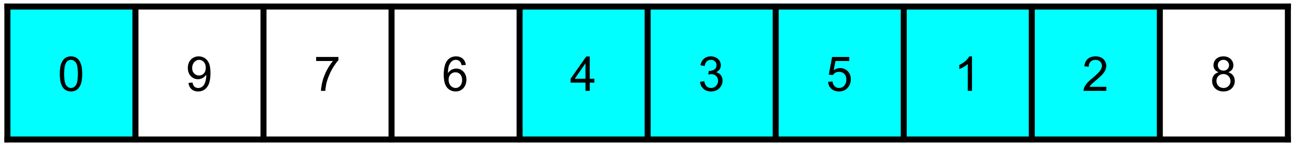
- step 2 is called the *partition* step
- consider the following list of unique elements

0	8	7	6	4	3	5	1	2	9

- assume that the pivot is 6

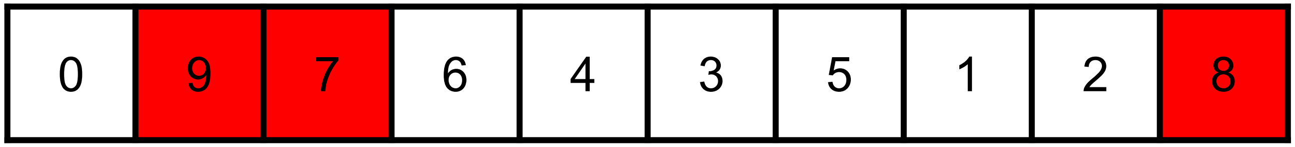
Quicksort

- the partition step reorders the list so that:
  - values less than the pivot are located before the pivot
    - we need to move the cyan elements before the pivot



0	9	7	6	4	3	5	1	2	8

  - values greater than the pivot are located after the pivot
    - we need to move the red elements after the pivot



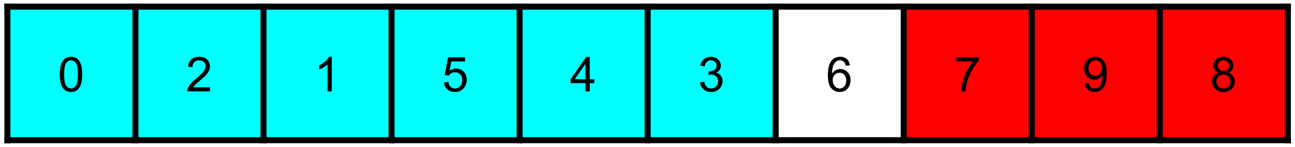
0	9	7	6	4	3	5	1	2	8

Quicksort

- can you describe an algorithm to perform the partitioning step?
  - talk amongst yourselves here

Quicksort

- after partitioning the list looks like:

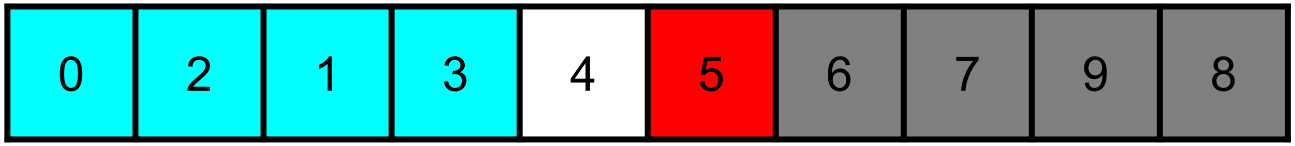


0	2	1	5	4	3	6	7	9	8

- partioning has 3 results:
  - the pivot is in its correct final sorted location
  - the left sublist contains only elements less than the pivot
  - the right sublist contains only elements greater than the pivot

Quicksort

- after partitioning we recursively quicksort the left sublist
- for the left sublist, let's assume that we choose 4 as the pivot
  - after partitioning the left sublist we get:

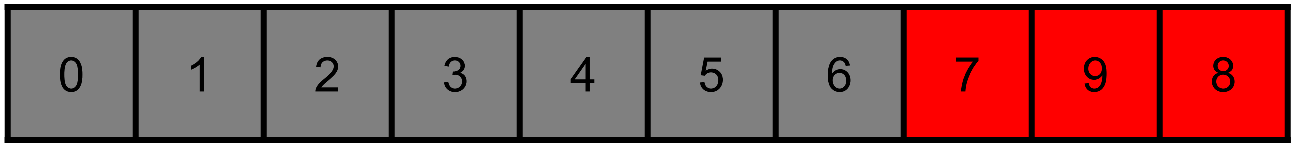


0	2	1	3	4	5	6	7	9	8

  - we then recursively quicksort the left and right sublists
    - □ and so on...

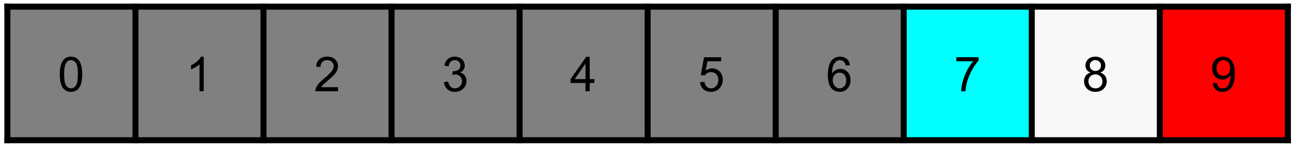
Quicksort

- eventually, the left sublist from the first pivoting operation will be sorted; we then recursively quicksort the right sublist:



0	1	2	3	4	5	6	7	9	8

- if we choose 8 as the pivot and partition we get:



0	1	2	3	4	5	6	7	8	9

- the left and right sublists have size 1 so there is nothing left to do

Quicksort

- the computational complexity of quicksort depends on:
  - the computational complexity of the partition operation
    - without proof I claim that this is $O(n)$ for a list of size n
  - how the pivot is chosen

Quicksort

- let's assume that when we choose a pivot we always choose the smallest (or largest) value in the sublist
  - yields a sublist of size $(n-1)$ which we recursively quicksort
- let $T(n)$ be the number of operations needed to quicksort a list of size n when choosing a pivot as described above
  - then the recurrence relation is:

$$T(n) = T(n-1) + O(n) \qquad \text{same as selection sort}$$

  - solving the recurrence results in

$$T(n) = O(n^2)$$

Quicksort

- let's assume that when we choose a pivot we always choose the median value in the sublist
  - yields 2 sublists of size $\left(\frac{n}{2}\right)$ which we recursively quicksort
- let $T(n)$ be the number of operations needed to quicksort a list of size n when choosing a pivot as described above
  - then the recurrence relation is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

same as merge sort

  - solving the recurrence results in

$$T(n) = O(n \log_2 n)$$

- what is the fastest way to sort a deck of playing cards?
- what is the big-O complexity?
- talk amongst ourselves here….

Proving correctness and terminaton

Proving Correctness and Termination

- to show that a recursive method accomplishes its goal you must prove:
  1. that the base case(s) and the recursive calls are correct
  2. that the method terminates

Proving Correctness

- to prove correctness:
  1. prove that each base case is correct
  2. assume that the recursive invocation is correct and then prove that each recursive case is correct

printItToo

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

Correctness of printItToo

1. (prove the base case) If **`n == 0`** nothing is printed; thus the base case is correct.
2. Assume that **`printItToo(s, n-1)`** prints the string **`s`** exactly **`(n – 1)`** times. Then the recursive case prints the string **`s`** exactly **`(n – 1)+1 = n`** times; thus the recursive case is correct.

Proving Termination

- to prove that a recursive method terminates:
  1. define the size of a method invocation; the size must be a non-negative integer number
  2. prove that each recursive invocation has a smaller size than the original invocation

Termination of printItToo

1. **`printItToo(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printItToo(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printItToo(s, n-1)`** is **`n-1`** (by definition) which is smaller than the original size **n**.

countZeros

```
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

Correctness of countZeros

1. (base cases) If the number has only one digit then the method returns **`1`** if the digit is zero and **`0`** if the digit is not zero; therefore, the base case is correct.
2. (recursive cases) Assume that **`countZeros(n/10L)`** is correct (it returns the number of zeros in the first **`(d – 1)`** digits of **`n`**).

   There are two recursive cases:

Correctness of countZeros

a. If the last digit in the number is zero, then the recursive case returns **`1 +`** the number of zeros in the first **`(d – 1)`** digits of **`n`**, which is correct.

b. If the last digit in the number is one, then the recursive case returns the number of zeros in the first **`(d – 1)`** digits of **`n`**, which is correct.

Termination of countZeros

1. Let the size of **`countZeros(n)`** be **`d`** the number of digits in the number **`n`**.
2. The size of the recursive invocation **`countZeros(n/10L)`** is **`d-1`**, which is smaller than the size of the original invocation.

Selection Sort

```
public class Recursion {

  // minToFront not shown

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

Prove that selection sort is correct and terminates.

```
}
```

Proving Termination

- prove that the algorithm on the next slide terminates

```
public class Print {

  public static void done(int n) {
    if (n == 1) {
      System.out.println("done");
    }
    else if (n % 2 == 0) {
      System.out.println("not done");
      Print.done(n / 2);
    }
    else {
      System.out.println("not done");
      Print.done(3 * n + 1);
    }
  }

}
```

Binary Search

- one reason that we care about sorting is that it is much faster to search a sorted list compared to sorting an unsorted list
- the classic algorithm for searching a sorted list is called *binary search*
- to search a list of size n for a value v:
    - look at the element e at index $\left(\frac{n}{2}\right)$
    - if $e > v$ recursively search the sublist to the left
    - if $e < v$ recursively search the sublist to the right
    - if $e == v$ then done

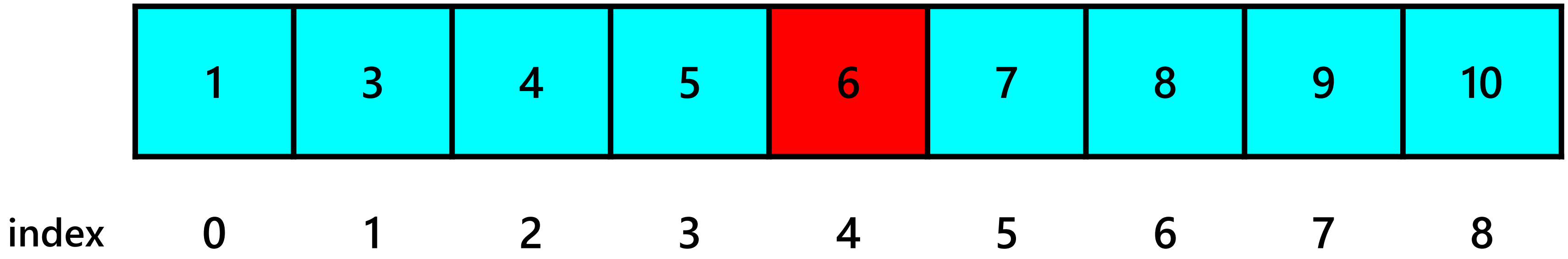
Binary Search

- consider the sorted list of size $n = 9$

	1	3	4	5	6	7	8	9	10
sublist index	0	1	2	3	4	5	6	7	8

Binary Search

- search for $v = 3$



	1	3	4	5	🟥 6	7	8	9	10
index	0	1	2	3	4	5	6	7	8

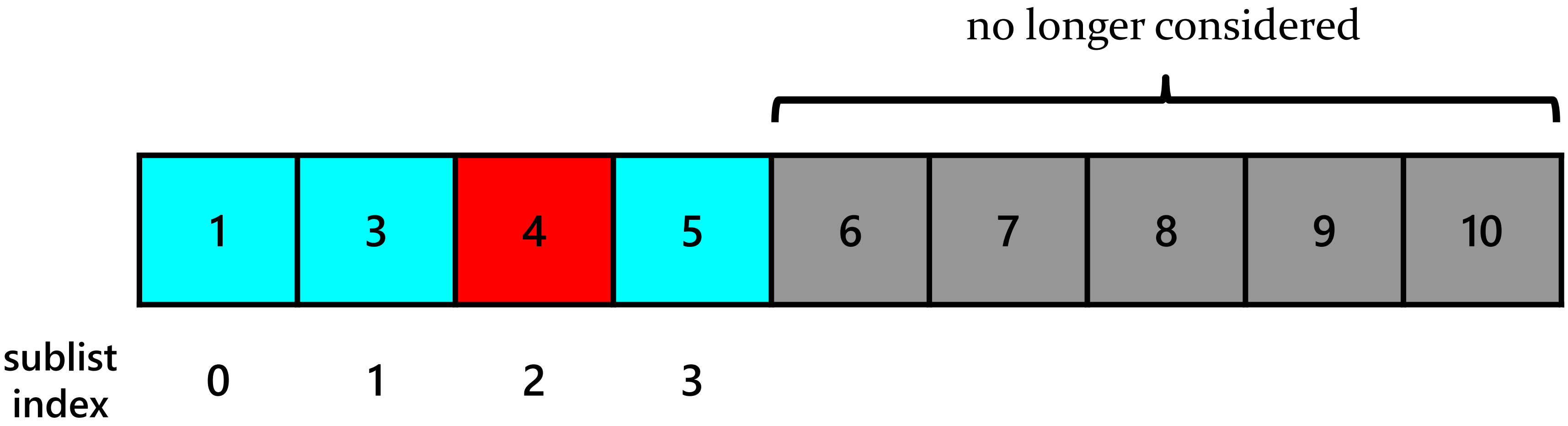
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 3$



no longer considered

	1	3	4	5	6	7	8	9	10
sublist index	0	1	2	3					

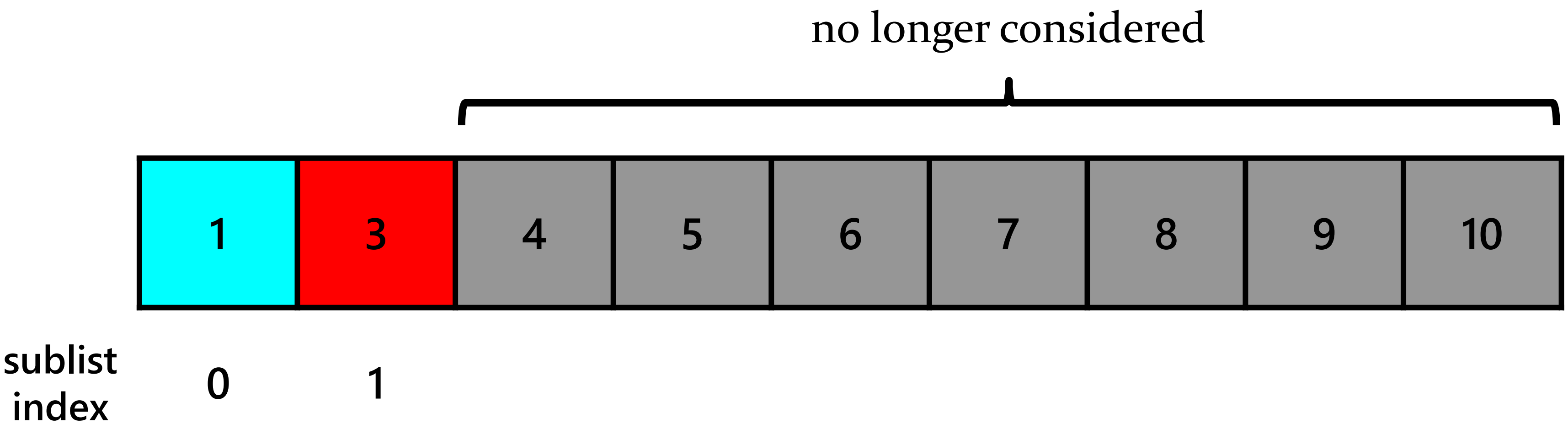
$$mid = \frac{4}{2} = 2$$

$$e = 4$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 3$



no longer considered

	1	3	4	5	6	7	8	9	10
sublist index	0	1							

$$mid = \frac{2}{2} = 1$$

$$e = 3$$

$$v == e, \text{done}$$

Binary Search

- search for $v = 2$

	1	3	4	5	6	7	8	9	10
sublist index	0	1	2	3	4	5	6	7	8

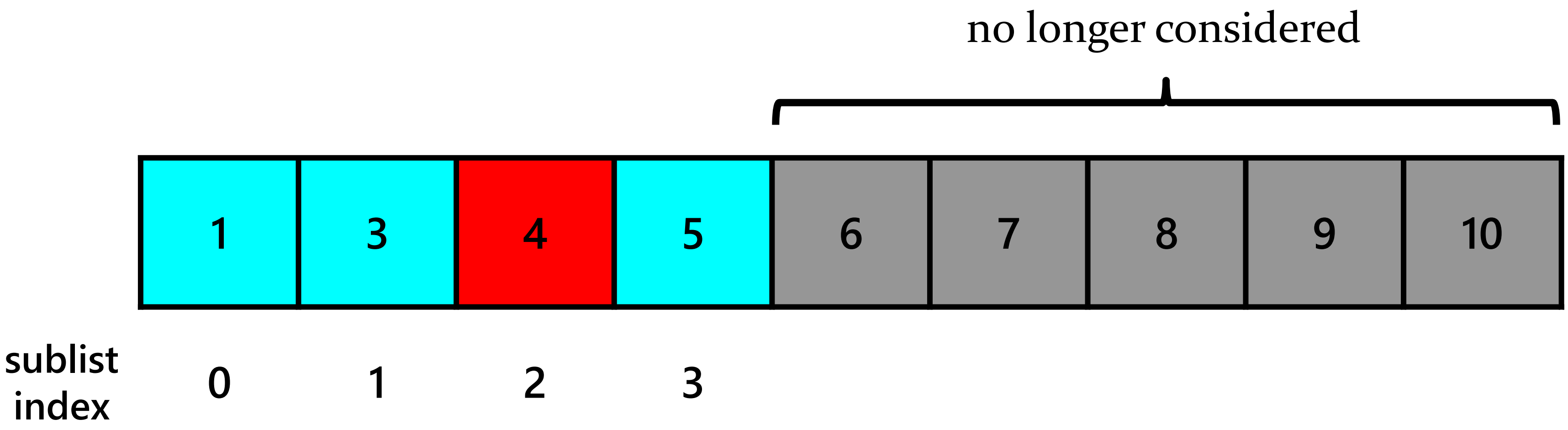
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



no longer considered

	1	3	4	5	6	7	8	9	10
sublist index	0	1	2	3					

$$mid = \frac{4}{2} = 2$$

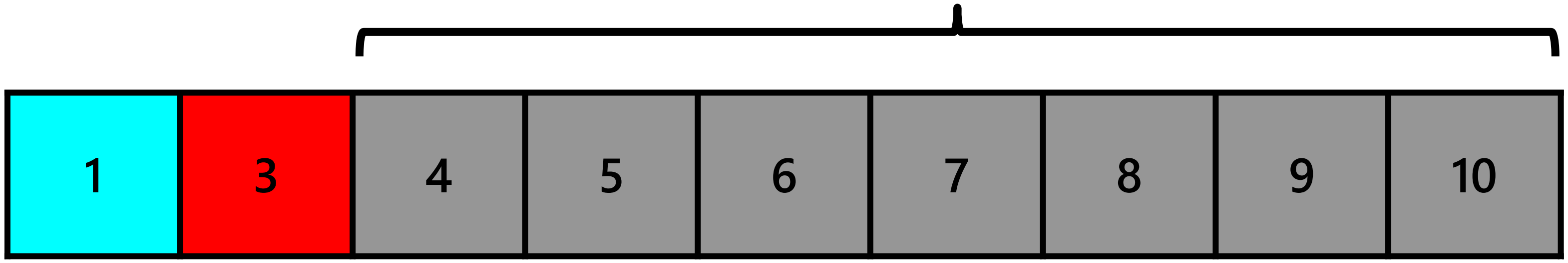
$$e = 4$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$

no longer considered



	1	3	4	5	6	7	8	9	10
sublist index	0	1							

$$mid = \frac{2}{2} = 1$$

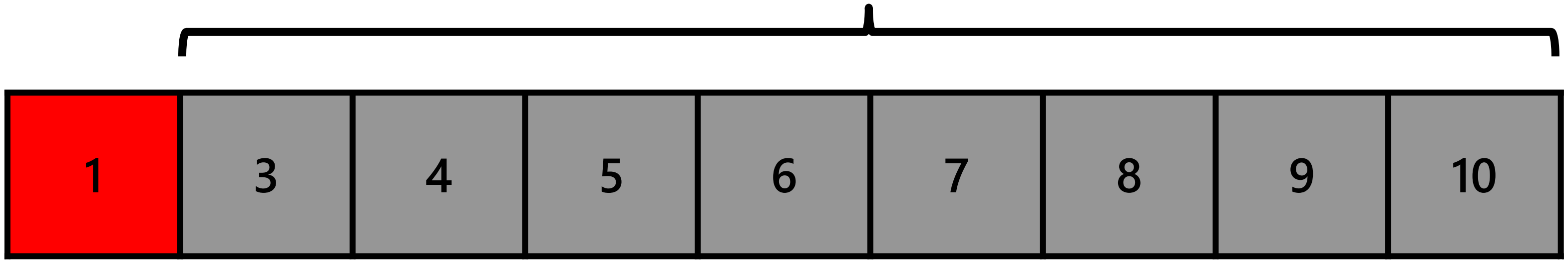
$$e = 3$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$

no longer considered



	1	3	4	5	6	7	8	9	10
sublist index	**0**								

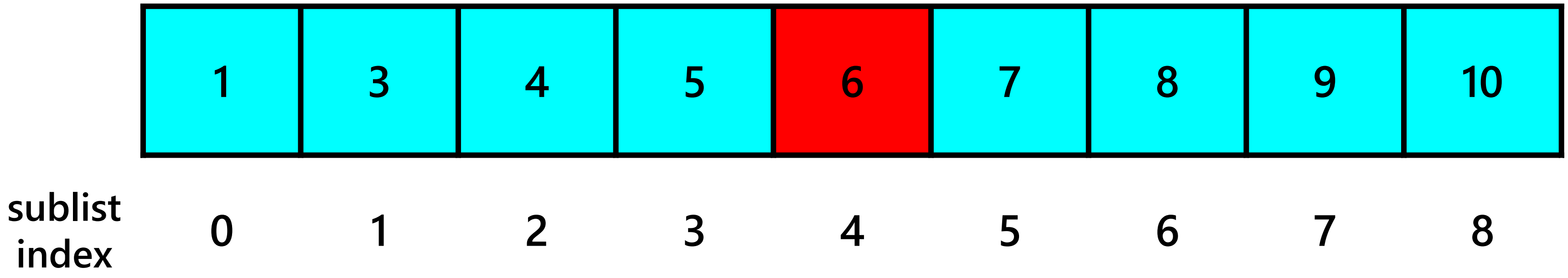
$$mid = \frac{1}{2} = 0$$

$$e = 1$$

$v > e$, recursively search the right sublist; right sublist is empty, done

Binary Search

- search for v =9



	1	3	4	5	6	7	8	9	10
sublist index	0	1	2	3	4	5	6	7	8

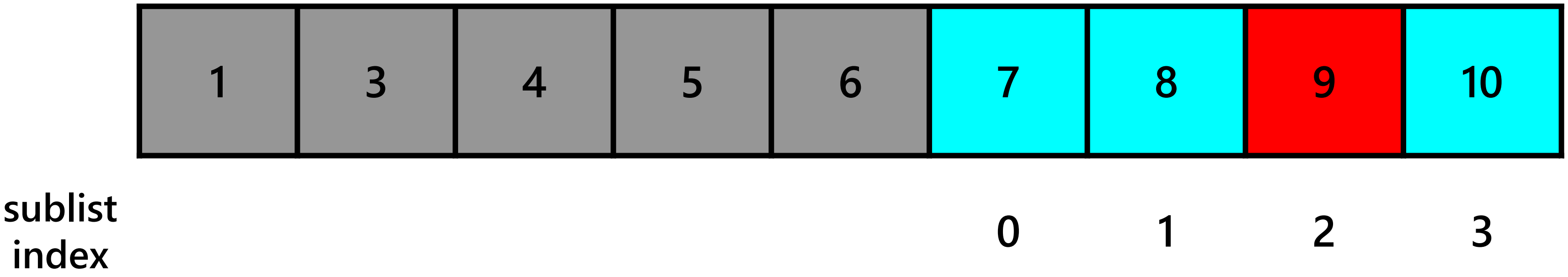
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v > e$, recursively search the right sublist

Binary Search

- search for v =9



	1	3	4	5	6	7	8	9	10
sublist index						0	1	2	3

$$mid = \frac{4}{2} = 2$$

$$e = 9$$

$$v == e, \text{done}$$

```java
/**
 * Searches a sorted list of integers for a given value using binary search.
 *
 * @param v the value to search for
 * @param t the list to search
 * @return true if v is in t, false otherwise
 */
public static boolean contains(int v, List<Integer> t) {
  if (t.isEmpty()) {
    return false;
  }
  int mid = t.size() / 2;
  int e = t.get(mid);
  if (e == v) {
    return true;
  }
  else if (v < e) {
    return Recursion.contains(v, t.subList(0, mid));
  }
  else {
    return Recursion.contains(v, t.subList(mid + 1, t.size()));
  }
}
```

Binary Search

- what is the recurrence relation?
- what is the big-O complexity?

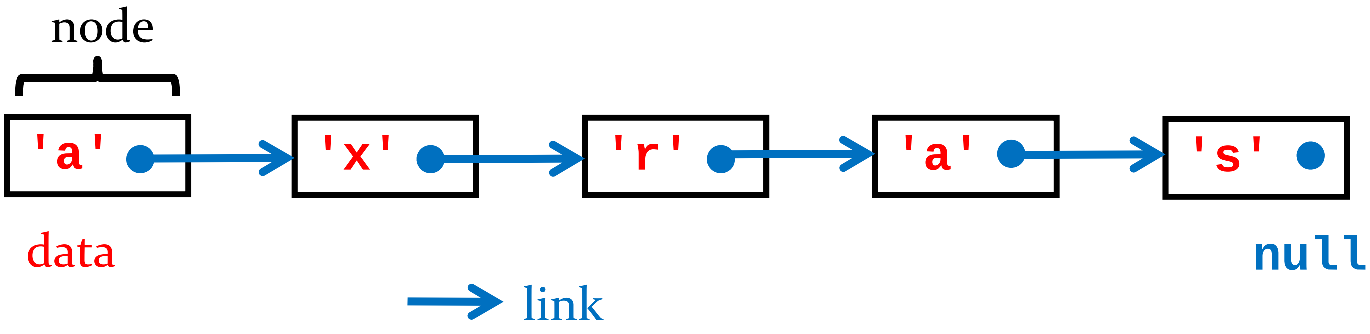
Revisiting Linked List

Recursive Objects

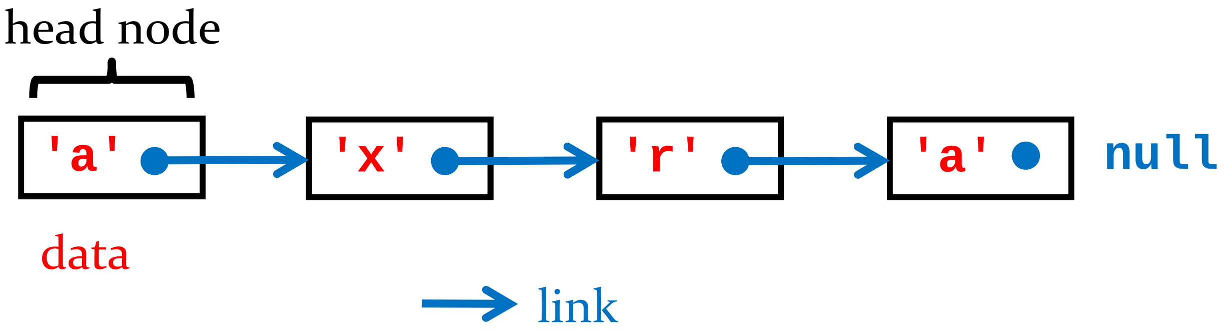
- an object that holds a reference to its own type is a recursive object
  - linked lists and trees are classic examples in computer science of objects that can be implemented recursively

Singly Linked List

- a data structure made up of a sequence of nodes
- each node has
  - some data
  - a field that contains a reference (a *link*) to the **next** node in the sequence
- suppose we have a linked list that holds characters; a picture of our linked list would be:



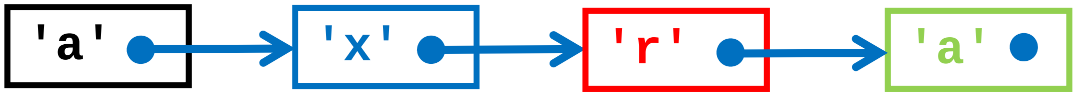
Singly Linked List



- the first node of the list is called the *head* node

Linked List

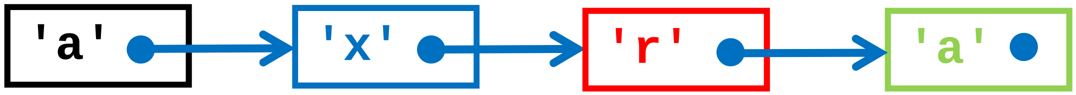
- each node can be thought of as the head of a smaller list



head of
`['a', 'x', 'r', 'a']`

Linked List

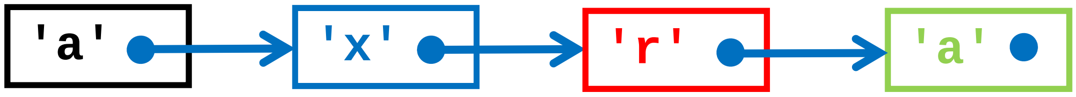
- each node can be thought of as the head of a smaller list



head of
`['x', 'r', 'a']`

Linked List

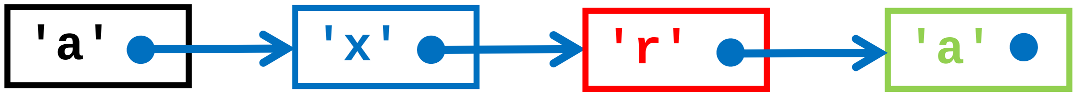
- each node can be thought of as the head of a smaller list



head of
['r', 'a']

Linked List

- each node can be thought of as the head of a smaller list



head of
`['a']`

Linked List

- the recursive structure of the linked list suggests that algorithms that operate on a linked list can be implemented recursively
  - e.g., **`getNode(int index)`** from Lab 5

```
/**
 * Returns the node at the given index.
 *
 * <p>
 * NOTE: This method is extremely useful for implementing many of the methods
 * of this class, but students should try to use this method only once in each
 * method.
 *
 * <p>
 * NOTE: This method causes a privacy leak and would not normally be
 * part of the public API; however, it is useful for testing purposes.
 *
 * @param index
 *           the index of the node
 * @return the node at the given index
 * @throws IndexOutOfBoundsException
 *            if index is less than 0 or greater than or equal the size of this
 *            list
 */
public Node getNode(int index) {
    this.checkIndex(index);
    return LinkedIntList.getNodeImpl(this.head, index);  // private static method
}
```

```java
/**
 * Returns the node located at the specified index in the
 * list with specified head node.
 *
 * @param head the head node of a linked list
 * @param index the index of the element
 * @return the node located index elements from the specified node
 */
private static Node getNodeImpl(Node head, int index) {
```

- base case(s)?
- recursive case?
- precondition(s)?

```java
}
```

```
/**
 * Returns the node located at the specified index in the
 * list with specified head node.
 *
 * @param head the head node of a linked list
 * @param index the index of the element
 * @return the node located index elements from the specified node
 */
private static Node getNodeImpl(Node head, int index) {
  if (index == 0) {
    return head;
  }
  return LinkedIntList.getNodeImpl(head.getNext(), index - 1);
}
```

Linked List

- recursive version of **`contains`**

```java
/**
 * Returns true if this list contains the specified element,
 * and false otherwise.
 *
 * @param elem the element to search for
 * @return true if this list contains the specified element,
 * and false otherwise
 */
public boolean contains(int elem) {
  if (this.size == 0) {
    return false;
  }
  return LinkedIntList.contains(this.head, elem);
}
```

```
/**
 * Returns true if the linked list with the specified head node contains
 * the specified element, and false otherwise.
 *
 * @param head the head node
 * @param elem the element to search for
 * @return true if the linked list with the specified head node contains
 * the specified element, and false otherwise
 */
private static boolean contains(Node head, int elem) {
```

- base case(s)?
- recursive case?
- precondition(s)?

```
}
```

```
/**
 * Returns true if the linked list with the specified head node contains
 * the specified element, and false otherwise.
 *
 * @param head the head node
 * @param elem the element to search for
 * @return true if the linked list with the specified head node contains
 * the specified element, and false otherwise
 */
private static boolean contains(Node head, int elem) {
  if (head.getData() == elem) {
    return true;
  }
  if (head.getNext() == null) {
    return false;
  }
  return LinkedIntList.contains(head.getNext(), elem);
}
```