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`**:

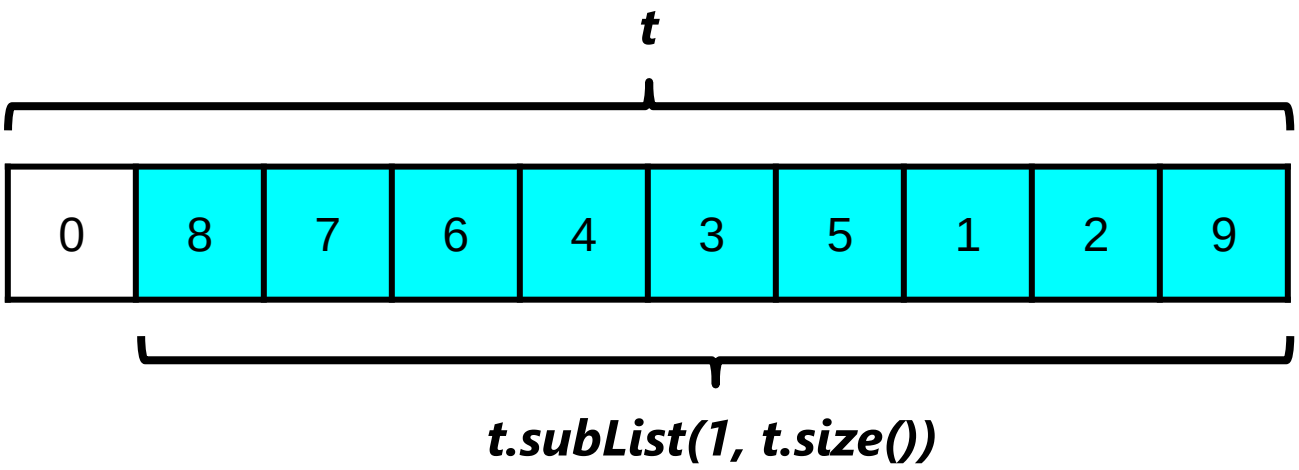
```
List<E> subList(int fromIndex, int toIndex)
```

Returns a 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

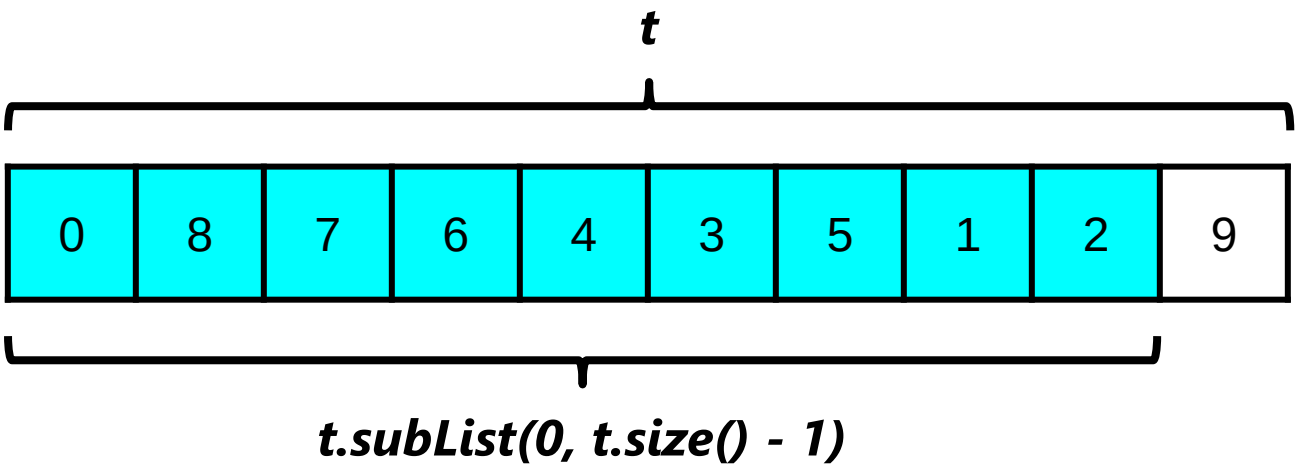
- the sub-list excluding the first element of the original list




```
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

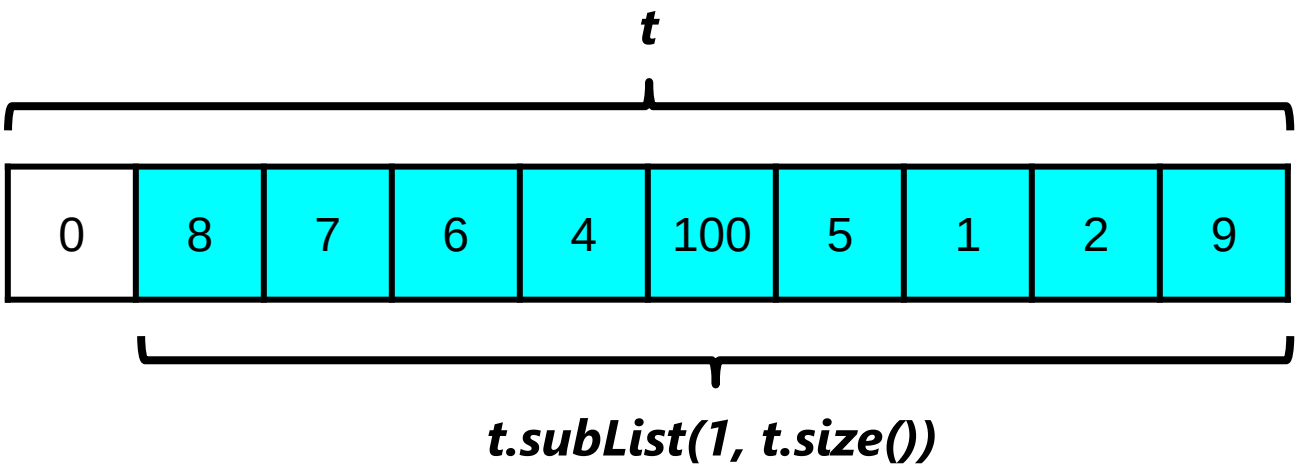
- the sub-list excluding the last element of the original list




```
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

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




```
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
```

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

```
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

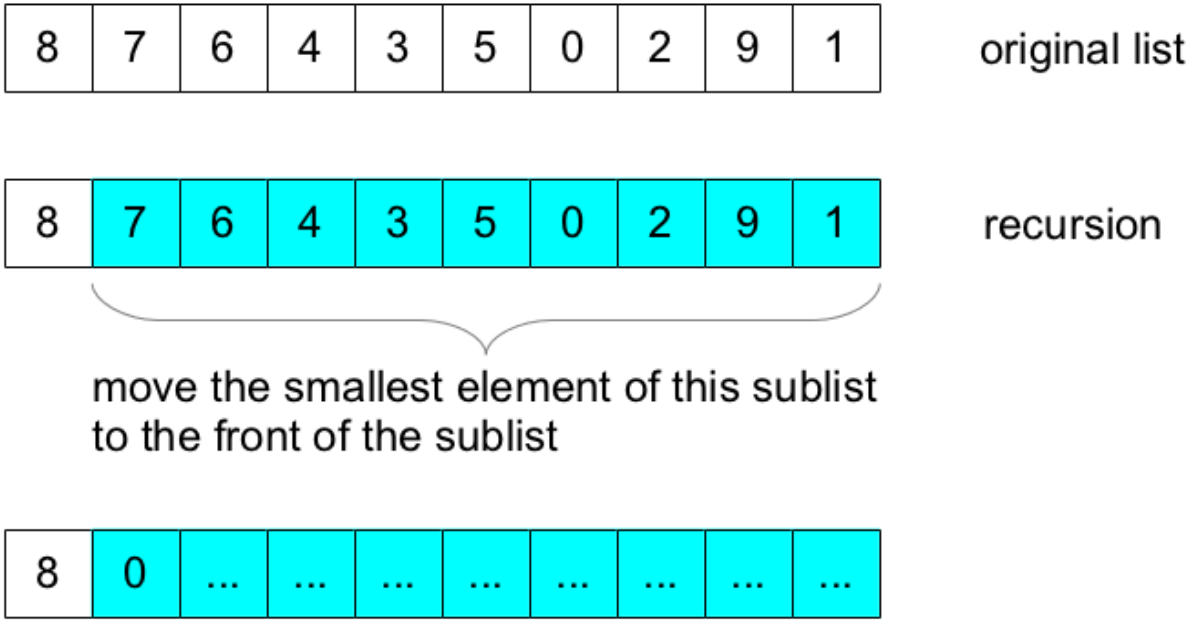
- recursive call?
  - to help deduce the recursive call assume that the method does exactly what its API says it does
    - e.g., **`contains(element, t)`** returns true if **`element`** is in the list **`t`** and false otherwise
  - use the assumption to write the recursive call or calls

```
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;
  }
}
```

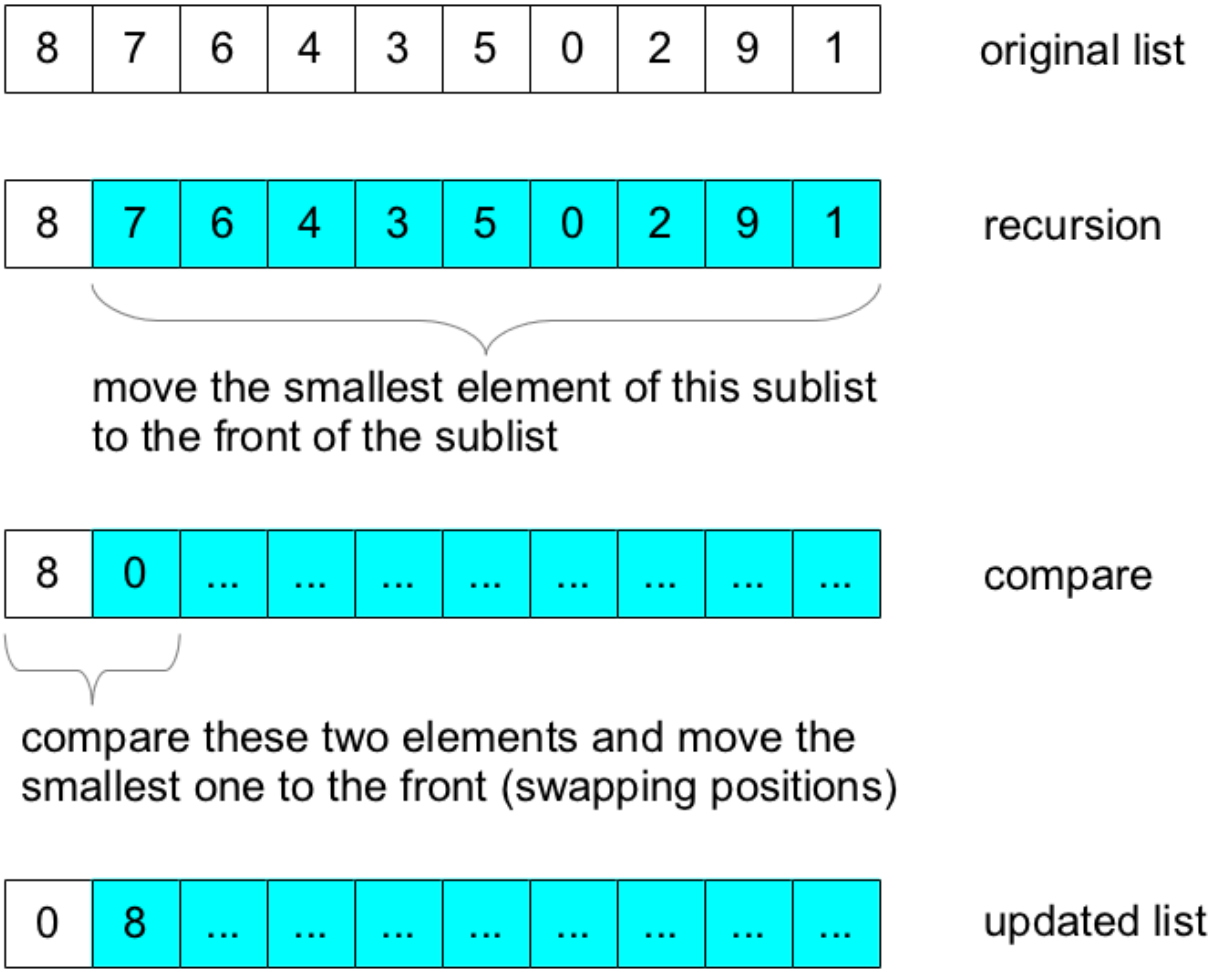
Recursion and Collections

- 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 Recursion {

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

  }
}
```

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()));

  }
}
```

Recursively Move Smallest to Front

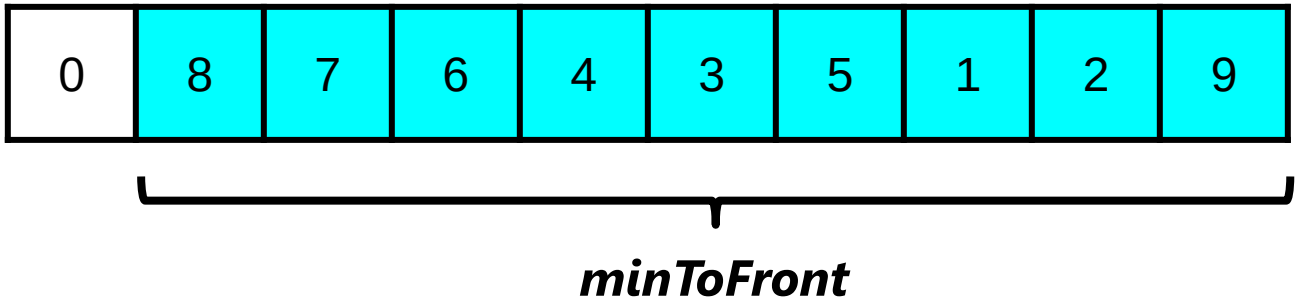
- compare and update?

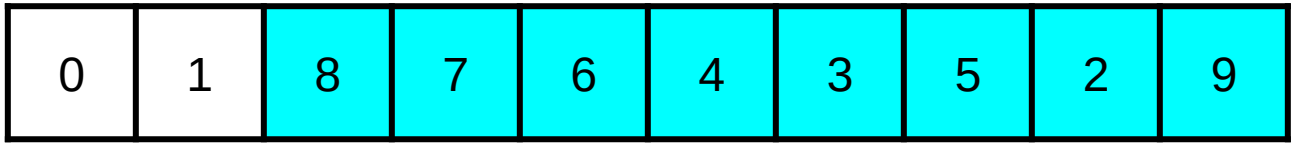
Recursively Move Smallest to Front

```
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);
    }
  }
}
```

Sorting the List

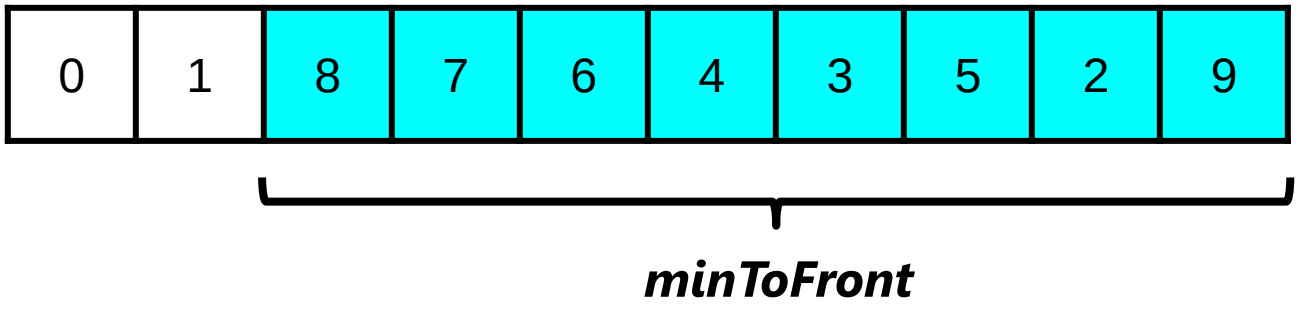
- observe what happens if you repeat the process with the sublist made up of the second through last elements:

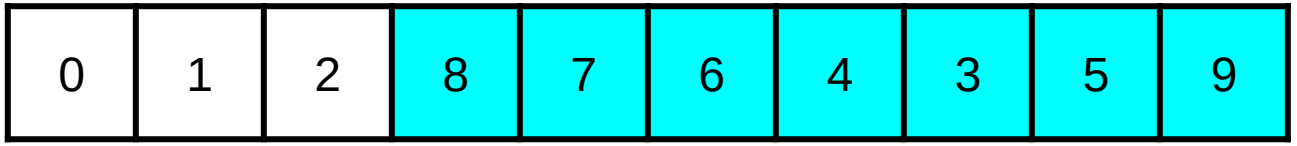




Sorting the List

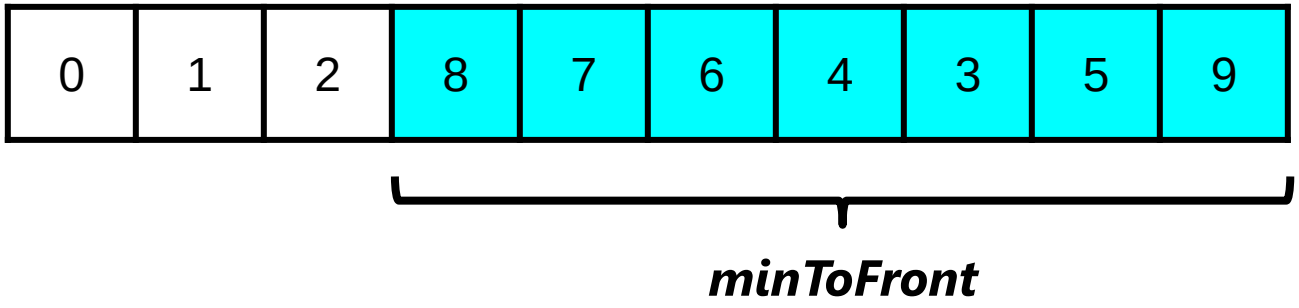
- observe what happens if you repeat the process with the sublist made up of the third through last elements:

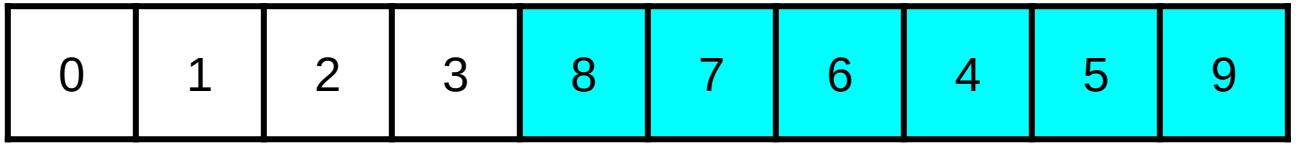




Sorting the List

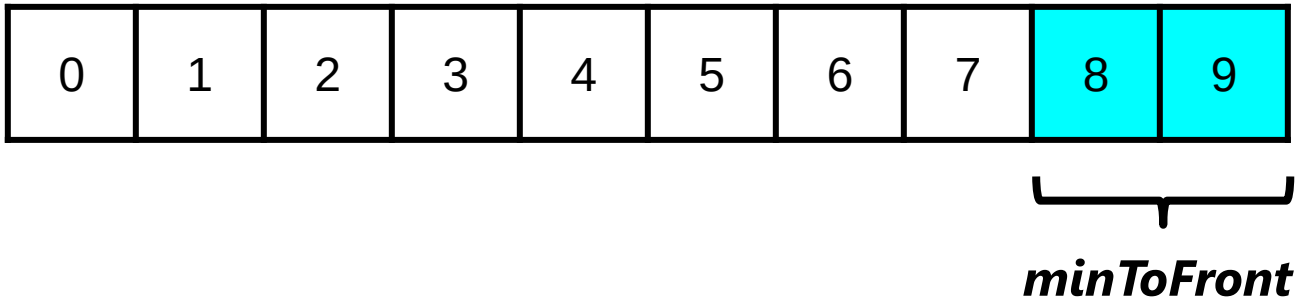
- observe what happens if you repeat the process with the sublist made up of the fourth through last elements:

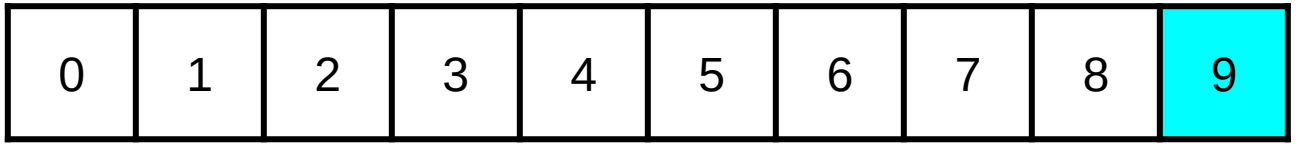




Sorting the List

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



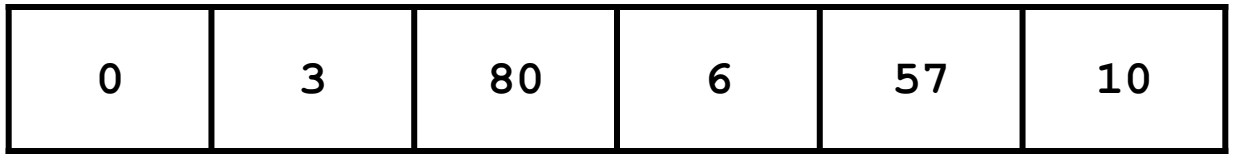


- this is the *selection sort* algorithm

Selection Sort

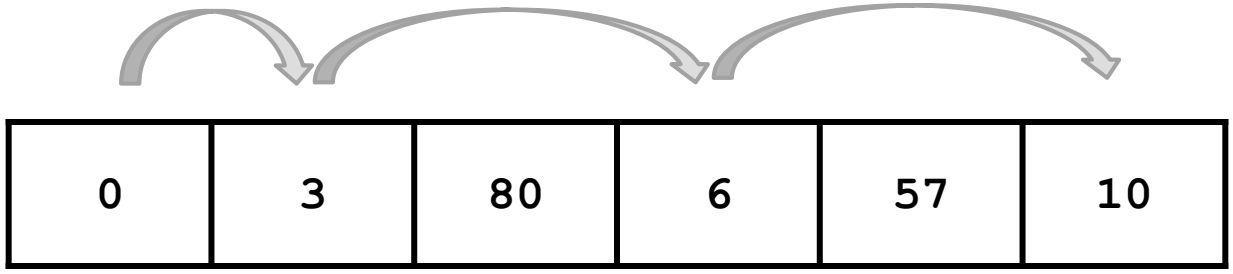
```
public class Recursion {

  // minToFront not shown

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


}
```

Jump It



- 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

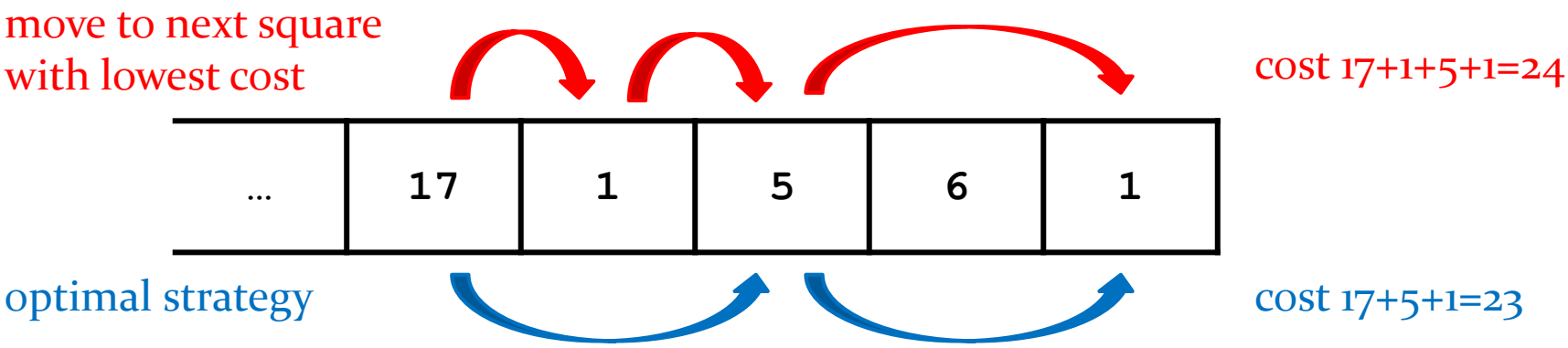


- solution for example:
  - move 1 square
  - move 2 squares
  - move 2 squares
    - total cost = 0 + 3 + 6 + 10 = 19

- 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

```
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

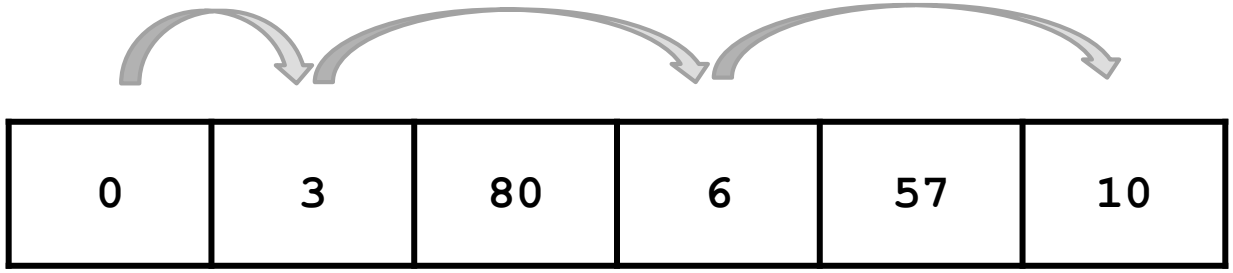
- return the smaller of the two costs

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



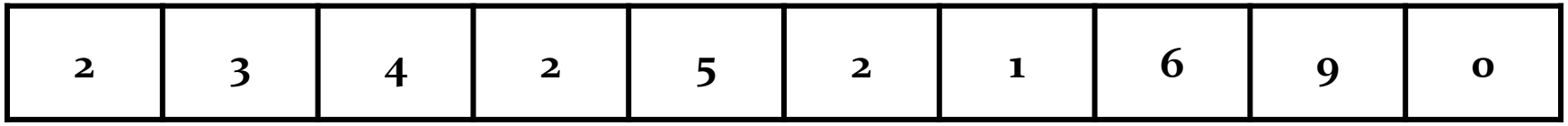
the method produces the list `[1, 2, 2]`

- consider using the following modified signature

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

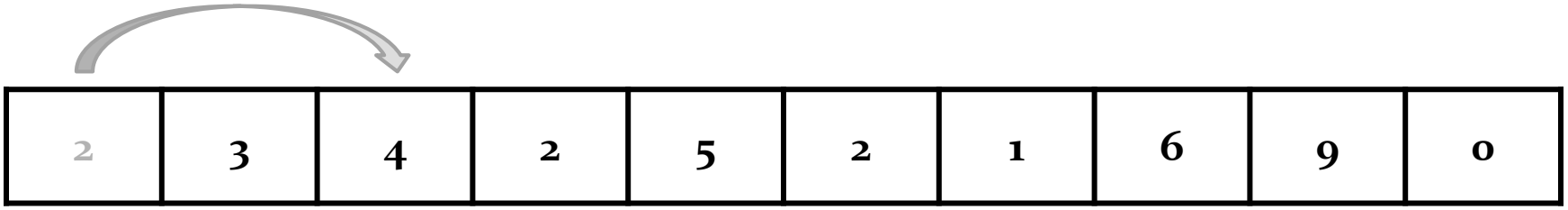
- the Jump It problem has a couple of nice properties:
  - the rules of the game make it impossible to move to the same square twice
  - the rules of the games make it impossible to try to move off of the board
- consider the following problem

- given a list of non-negative integer values:

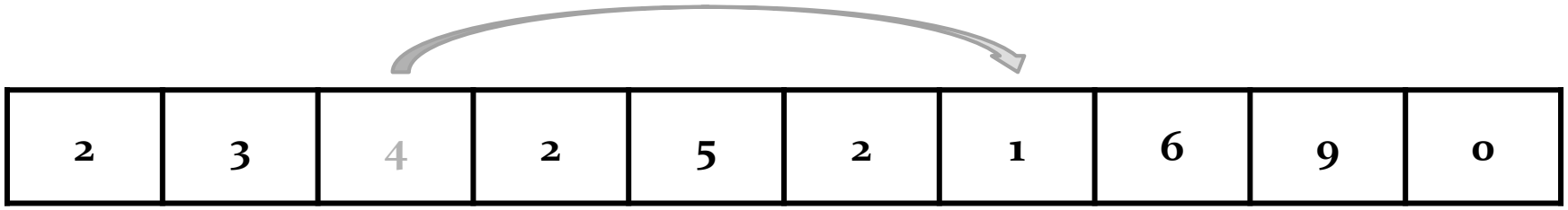


  - starting from the first element try to reach the last element (whose value is always zero)
  - you may move left or right by the number of elements equal to the value of the element that you are currently on
  - you may not move outside the bounds of the list

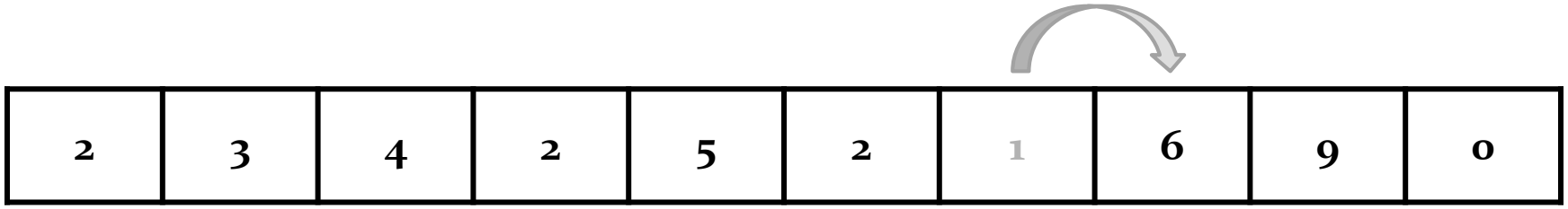
Solution 1



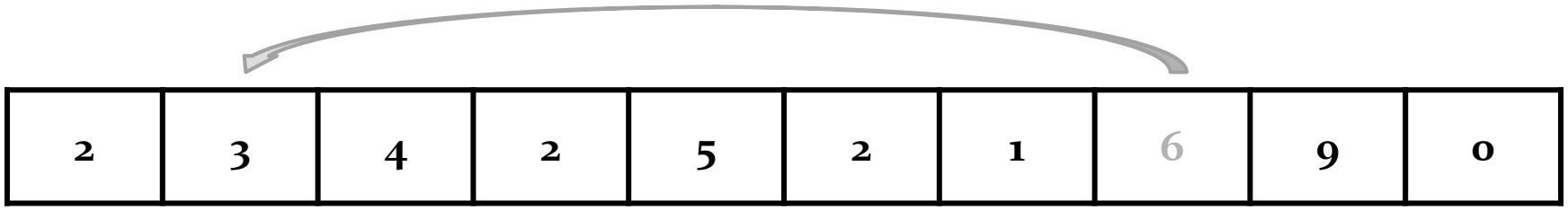
Solution 1



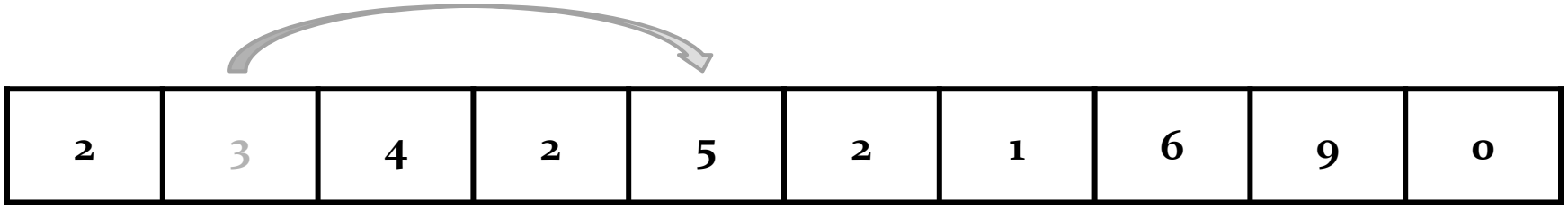
Solution 1



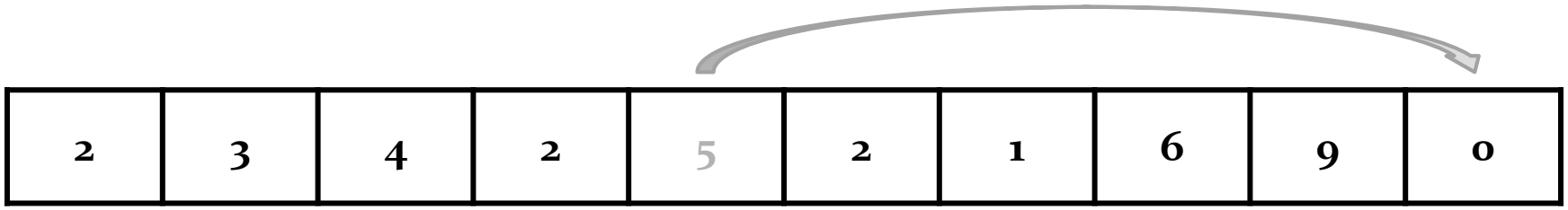
Solution 1



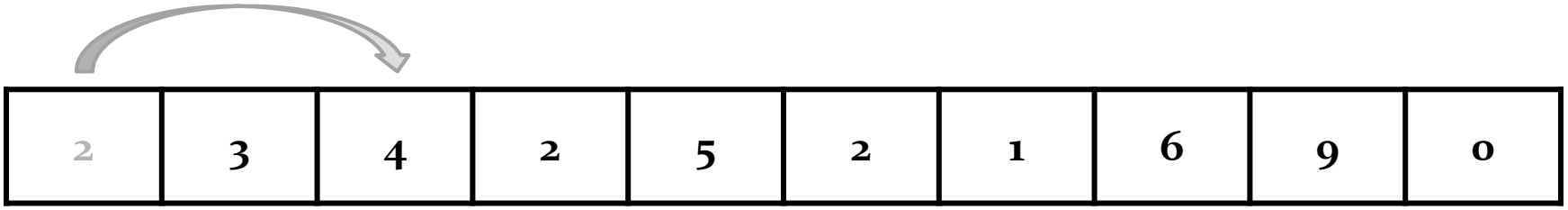
Solution 1



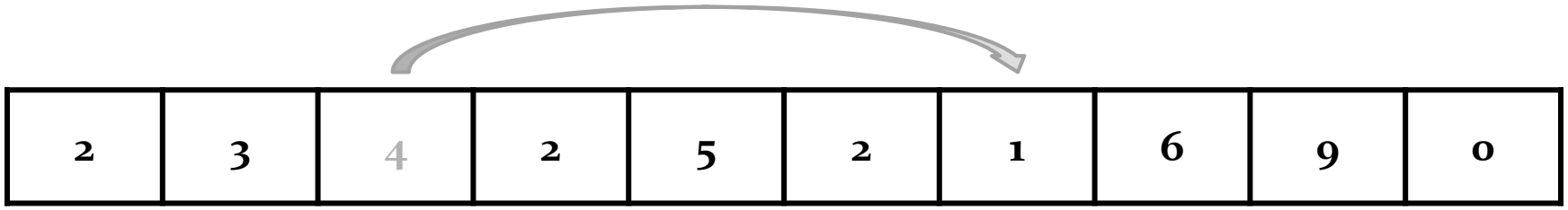
Solution 1



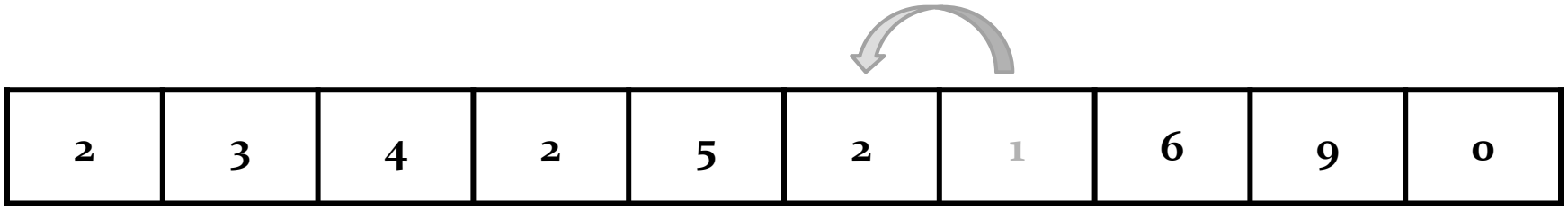
Solution 2



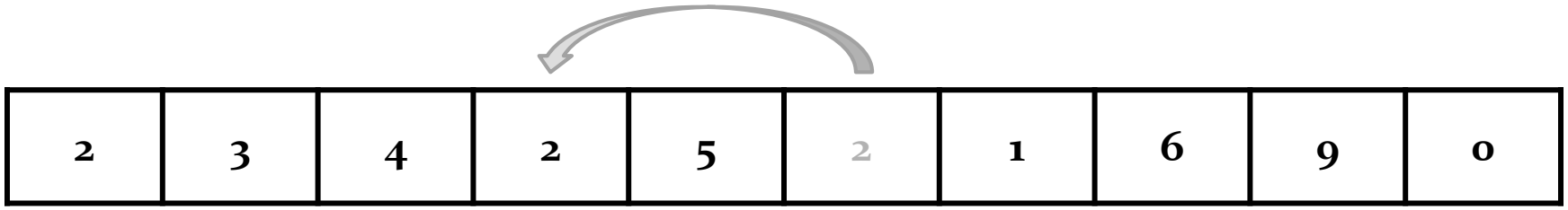
Solution 2



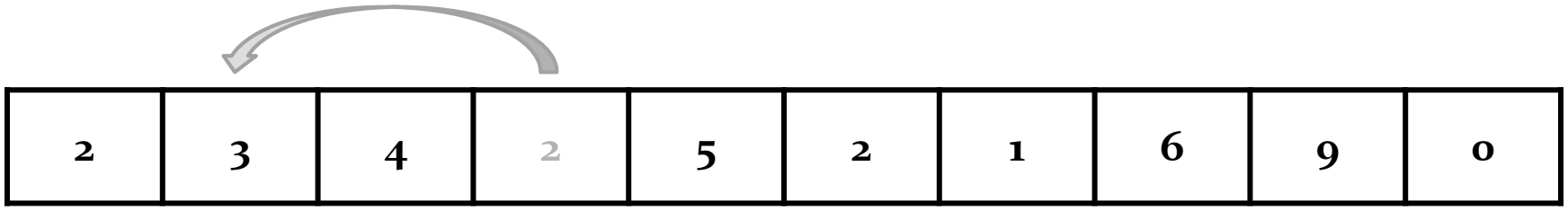
Solution 2



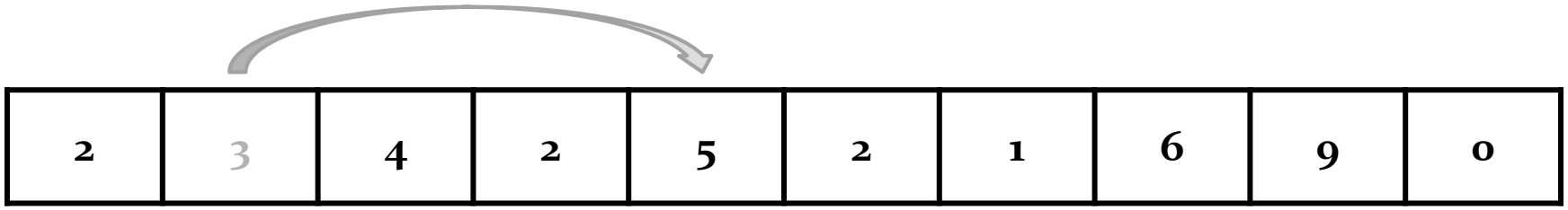
Solution 2



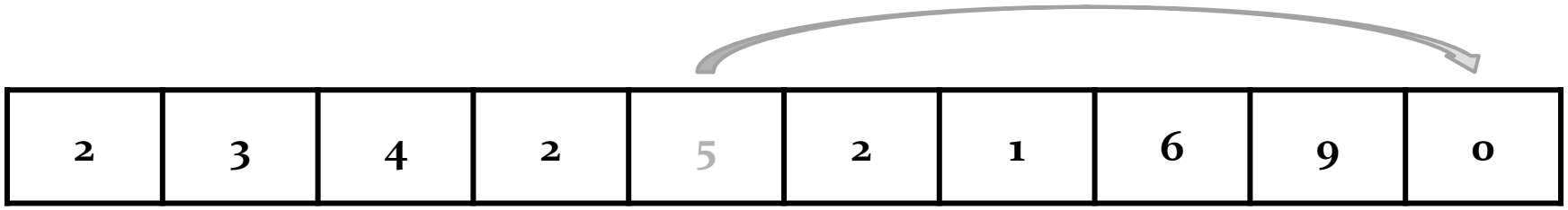
Solution 2



Solution 2

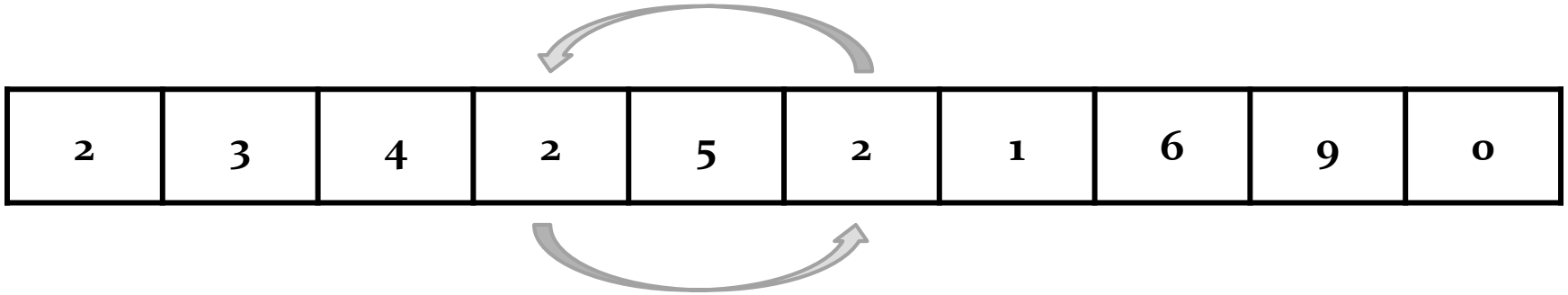


Solution 2



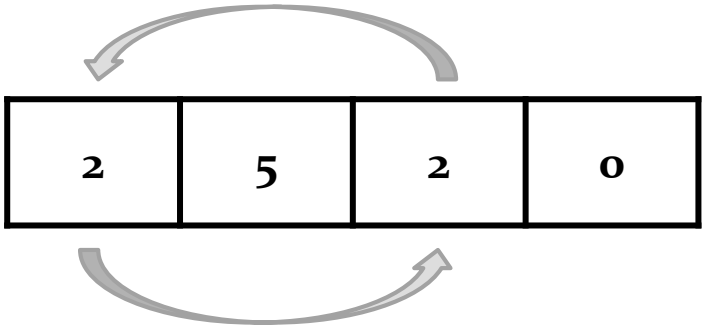
Cycles

- 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



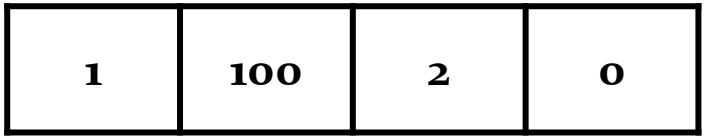
Cycles

- 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



- 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) {
}
```

```
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;




}
```

```
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);
    }
```

```
}
```

```
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()) {


    }

}
```

```
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);
    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...









}
```