

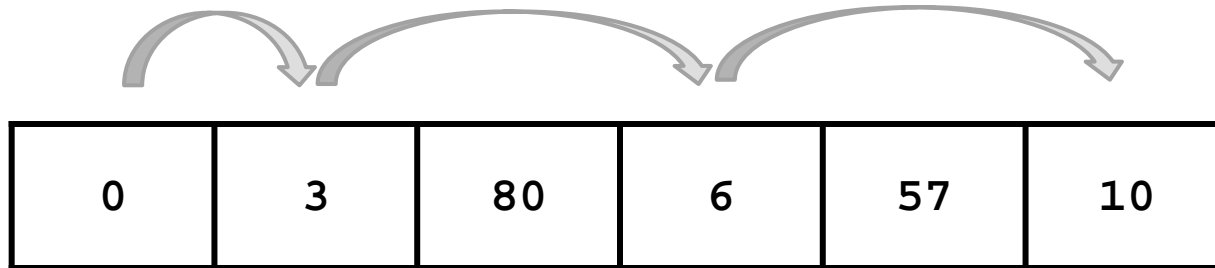
Recursion

Jump It

0	3	80	6	57	10
---	---	----	---	----	----

- ▶ board of n squares, $n \geq 2$
- ▶ 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

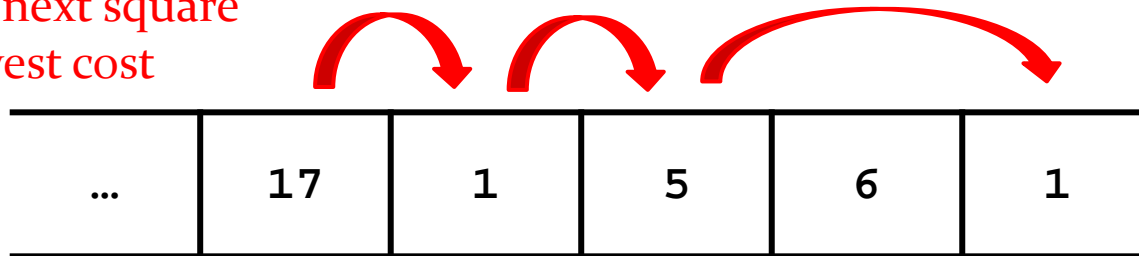
□ total cost = $0 + 3 + 6 + 10 = 19$

▶ 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

move to next square
with lowest cost



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);  
    }  
  
}
```

Jump It

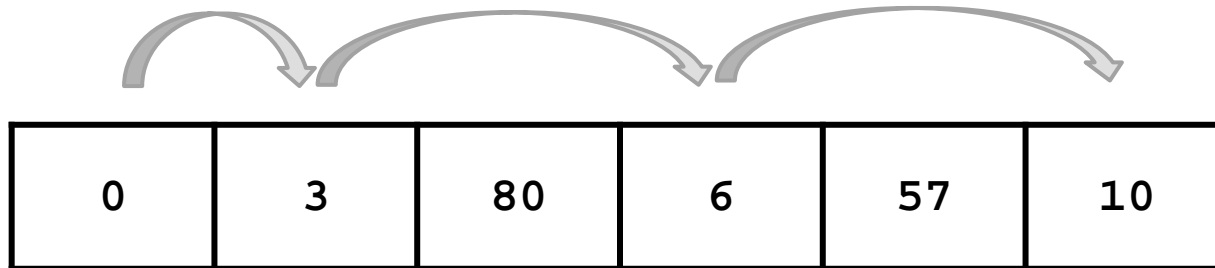
- ▶ recursive case(s):
 - ▶ compute the cost of moving 1 square
 - ▶ compute the cost of moving 2 squares
- ▶ return the smaller of the two costs

Jump It

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

Jump It

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



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

- ▶ consider using the following modified signature

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

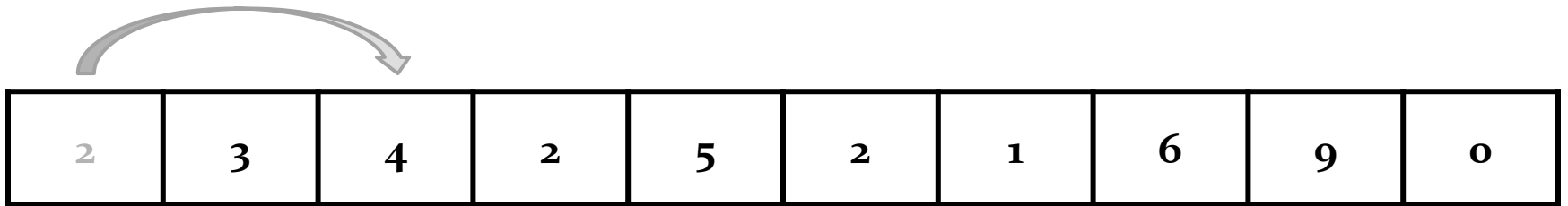
-
- ▶ the Jump It problem has a couple of nice properties:
 - ▶ the rules of the game make it impossible to move to the same square twice
 - ▶ the rules of the games make it impossible to try to move off of the board
 - ▶ consider the following problem

-
- ▶ given a list of non-negative integer values:

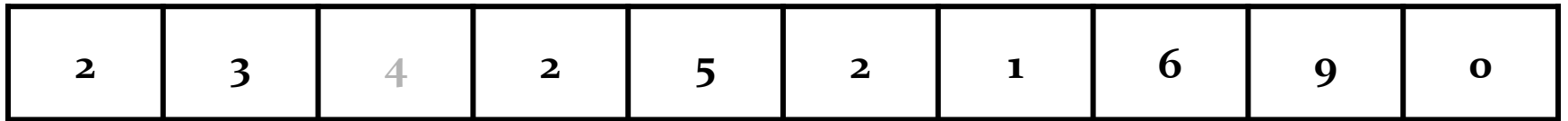
2	3	4	2	5	2	1	6	9	0
---	---	---	---	---	---	---	---	---	---

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

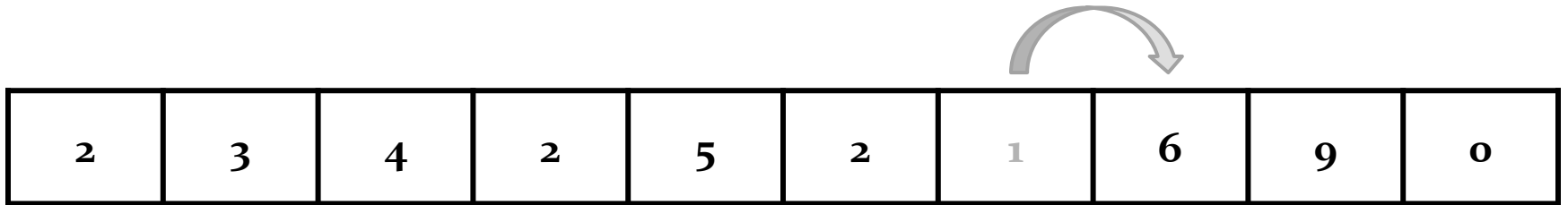
Solution 1



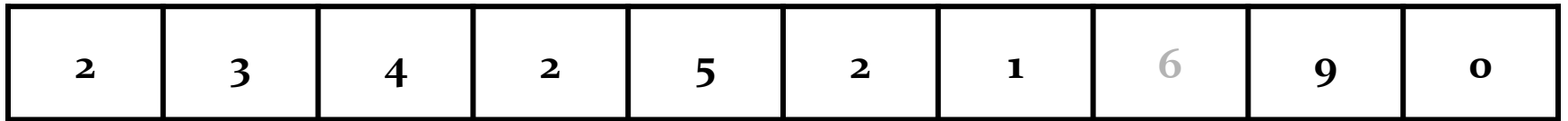
Solution 1



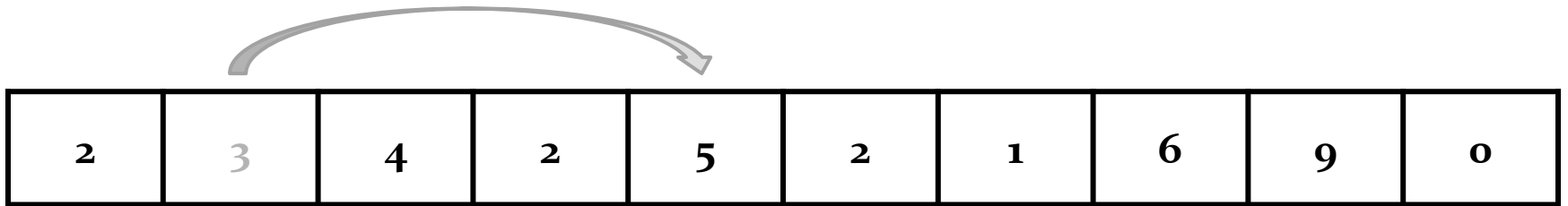
Solution 1



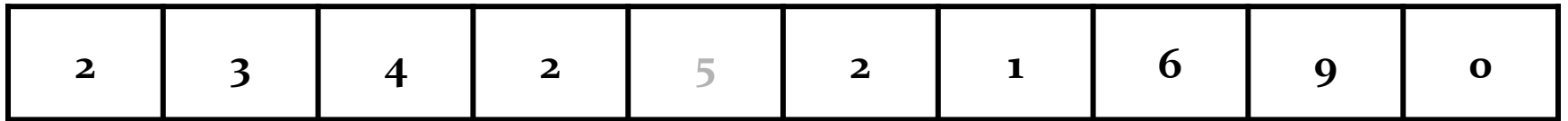
Solution 1



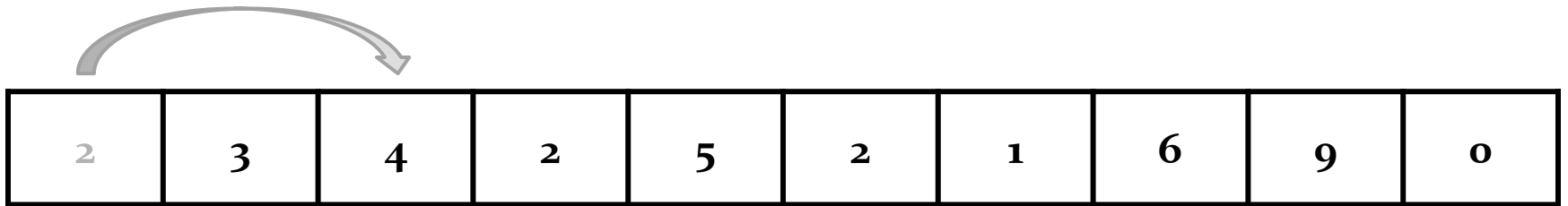
Solution 1



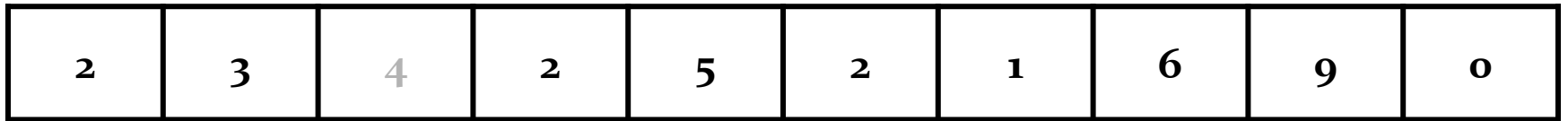
Solution 1



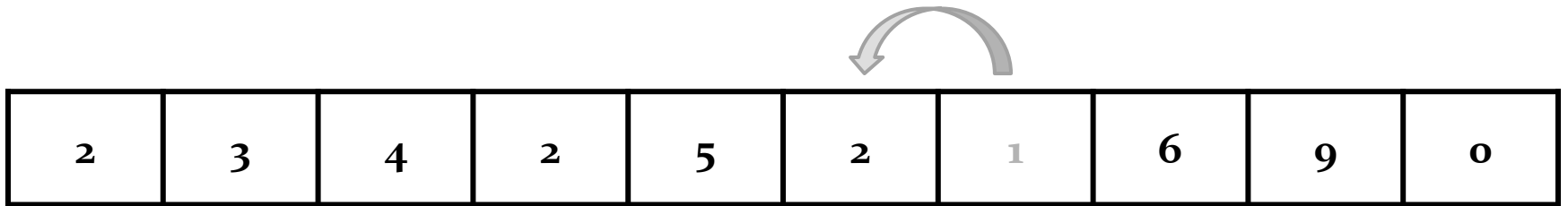
Solution 2



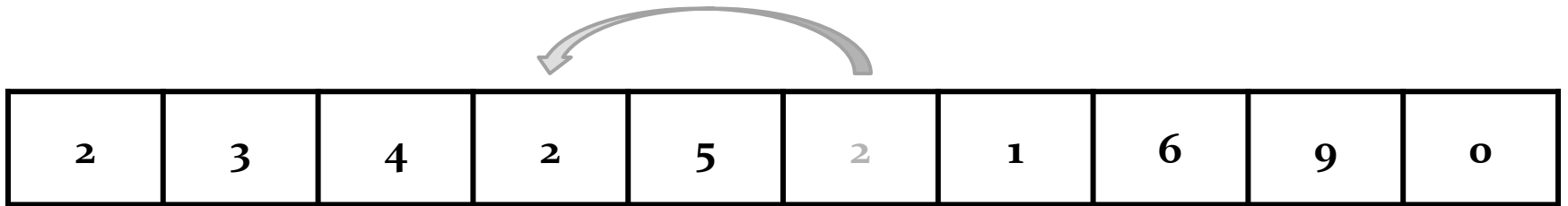
Solution 2



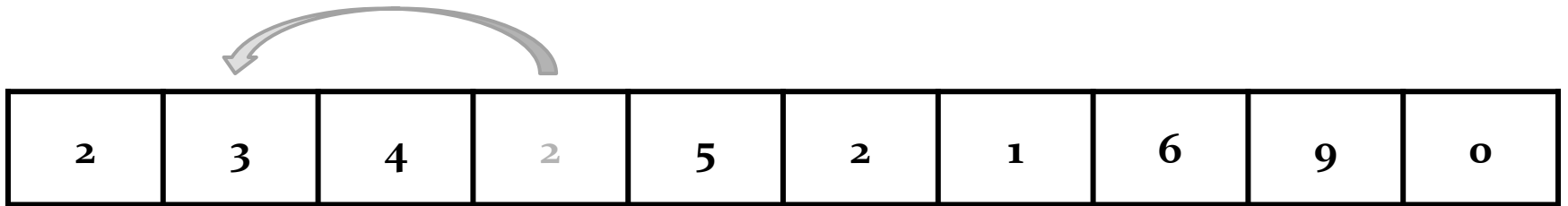
Solution 2



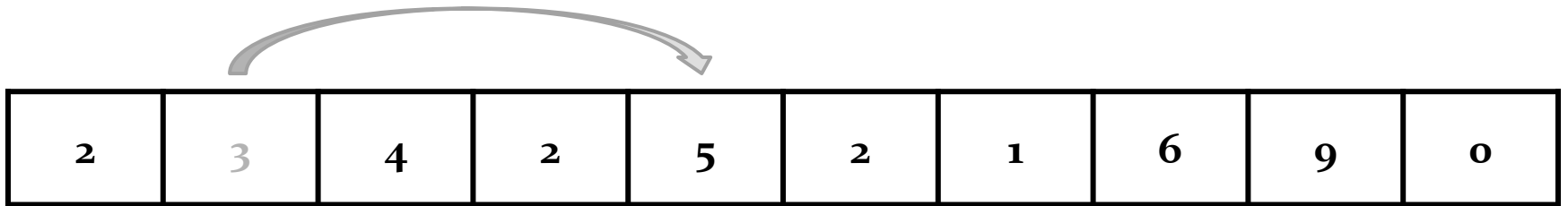
Solution 2



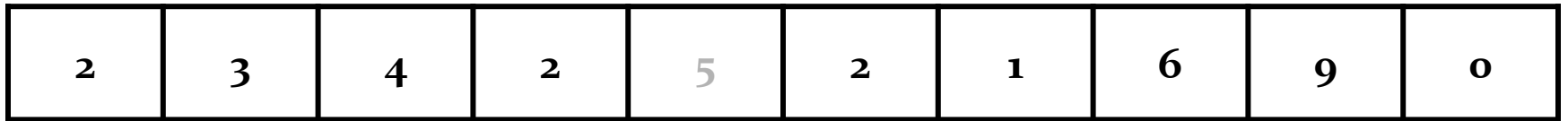
Solution 2



Solution 2

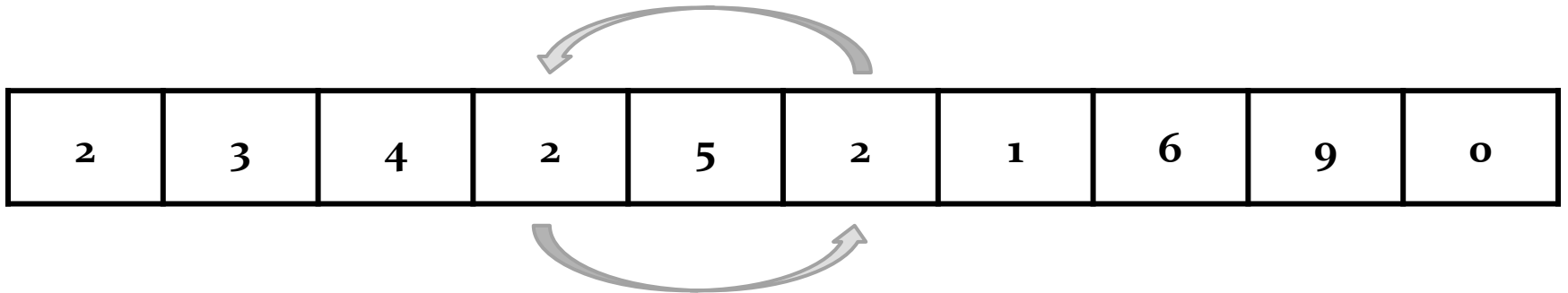


Solution 2



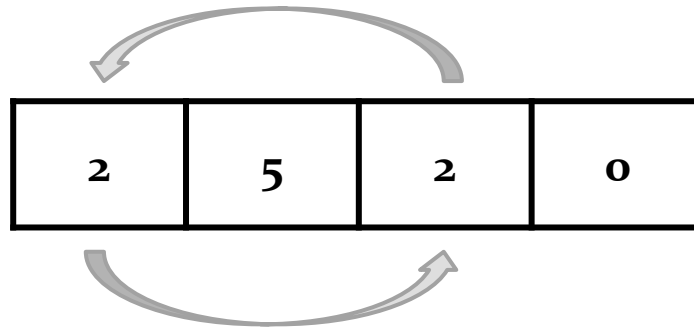
Cycles

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



Cycles

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



Cycles

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

2	1	2	2	2	2	2	6	3	0
---	---	---	---	---	---	---	---	---	---

No Solution

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

1	100	2	0
----------	------------	----------	----------

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

Recursion

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

```
/**
 * 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;  
    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;
}
```



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

