

**AP<sup>®</sup> COMPUTER SCIENCE AB  
2007 SCORING GUIDELINES**

**Question 1: Sliding Puzzle (Design)**

<b>Part A:</b>	isDone	<b>3 1/2 points</b>
----------------	--------	---------------------

- +1 1/2 2-D traversal
  - +1/2 correctly access an element of `values` in context of loop
  - +1 access all `side2` values (lose this if index out-of-bounds)
- +2 classify (must check `values`)
  - +1 return false if numbers out of order
  - +1 return true if numbers in order (ignoring 0)

<b>Part B:</b>	initialize	<b>4 1/2 points</b>
----------------	------------	---------------------

- +1 1/2 pick random element from `temp`
  - +1 generate random int in range `0..temp.size()`
  - +1/2 access element at the generated index
- +1 1/2 store element in `values`
  - +1 assign accessed element to any index
  - +1/2 correctly assign to empty index
- +1/2 `temp.remove(index)`
- +1 assign & remove all `side2` values

<b>Part C:</b>	$O(n^2)$	<b>1/2 point</b>
----------------	----------	------------------

<b>Part D:</b>	$O(n)$	<b>1/2 point</b>
----------------	--------	------------------

**AP<sup>®</sup> Computer Science AB  
2007 Canonical Solutions**

**Question 1: Sliding Puzzle (Design)**

**PART A:**

```
public boolean isDone()
{
    int nextAns = 1;
    for (int[] nextRow : values) {
        for (int nextVal : nextRow) {
            if (nextVal == nextAns) {
                nextAns++;
            }
            else if (nextVal != 0) {
                return false;
            }
        }
    }
    return true;
}
```

**PART B:**

```
public void initialize()
{
    ArrayList<Integer> temp = new ArrayList<Integer>();
    for (int j = 0; j < side*side; j++)
        temp.add(new Integer(j));

    for (int r = 0; r < side; r++) {
        for (int c = 0; c < side; c++) {
            int randIndex = (int)(Math.random()*temp.size());
            values[r][c] = temp.get(randIndex);
            temp.remove(randIndex);
        }
    }
}
```

**PART C:**

$O(n^2)$

**PART D:**

$O(n)$

- (a) Write the `SlidingPuzzle` method `isDone`. Method `isDone` returns `true` if the values in the puzzle appear in increasing order when traversed in row-major order; otherwise, it returns `false`. The value 0 (denoting the hole) may appear anywhere within the puzzle.

Complete method `isDone` below.

```

/** Precondition: the puzzle grid contains the distinct values 0 through side2 - 1
 * @return true if the tiles in the puzzle are all arranged in increasing order
 *         (the hole value 0 may be in any position);
 *         false otherwise
 */
public boolean isDone()
{
    for (int r = 0; r < side; r++)
        for (int c = 0; c < side - 1; c++)
        {
            if (values[r][c] > values[r][c+1]
                && values[r][c+1] != 0)
                return false;
        }
    return true;
}

```

Part (b) begins on page 6.

**GO ON TO THE NEXT PAGE.**

- (b) Write the `SlidingPuzzle` method `initialize`. Method `initialize` fills the `side × side` values `grid` with random integers 0 through  $side^2 - 1$ , without repeating numbers.

Your implementation must use the following algorithm.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through  $n = side^2 - 1$ . The code for this step is provided.

Starting with the first element of the `grid` values, repeat steps 2 and 3 until the `grid` has been filled.

2. Pick a random element from `temp` and place that element into the next empty `grid` location.
3. Remove that element from `temp` by calling the `ArrayList` `remove` method on its index.

Complete method `initialize` below. The method has been started for your convenience.

```

/** Initializes the puzzle by placing numbers 0 through  $side^2 - 1$  into random locations
 */
public void initialize()
{
    ArrayList<Integer> temp = new ArrayList<Integer>();
    for (int j = 0; j < side * side; j++)
        temp.add(new Integer(j));

    // Write your solution below.

    Random randNumGen = RandomGenerator.getInstance();

    for (int r=0; r < side; r++)
        for (int c=0; c < side; c++)
        {
            int randomPos = randNumGen.nextInt(temp.size());
            values[r][c] = temp.get(randomPos);
            temp.remove(randomPos);
        }
}
}
}

```

**GO ON TO THE NEXT PAGE.**

- (c) What is the expected big-Oh running time of the initialization algorithm described in part (b), in terms of  $n$ , the number of tiles?

Since step one of the initialization has a running time of  $O(N)$  and steps 2 and 3 have a worst case scenario of run time  $O(N^2)$  because of shifting elements after removing the first element.

$$O(N + N^2) = O(N^2)$$

⇒ The expected big-Oh running time of the initialization is  $O(N^2)$ .

- (d) Consider a variation of the algorithm described in part (b) in which Step 3 is changed.

1. Initialize an ArrayList<Integer> named temp with values 0 through  $n = \text{side}^2 - 1$ .

Starting with the first element of the grid values, repeat steps 2 and 3 until the grid has been filled.

2. Pick a random element from temp and place that element into the next empty grid location.
3. Replace that element with the element in the last index of temp, then remove the last element from temp. (Note: removing the last item of an ArrayList is a constant time operation.)

What is the expected big-Oh running time of this variation in terms of  $n$ , the number of tiles?

The big-Oh running time of this variation will be only  $O(N)$ .

Since we replace the element with element in the last index, the running time for removing is constant. There are no shifting involved.

Therefore,

1<sup>st</sup> step has a running time of  $O(N)$   
2<sup>nd</sup> & 3<sup>rd</sup> steps have a running time of  $O(N)$  too

$$O(N + N) = O(N)$$

**GO ON TO THE NEXT PAGE.**

- (a) Write the SlidingPuzzle method isDone. Method isDone returns true if the values in the puzzle appear in increasing order when traversed in row-major order; otherwise, it returns false. The value 0 (denoting the hole) may appear anywhere within the puzzle.

Complete method isDone below.

```

/** Precondition: the puzzle grid contains the distinct values 0 through side2 - 1
 * @return true if the tiles in the puzzle are all arranged in increasing order
 *         (the hole value 0 may be in any position);
 *         false otherwise
 */
public boolean isDone()

```

```

    int last = 0;
    for (int r = 0; r < side; r++)
        for (int c = 0; c < side; c++)
            if (values[r][c] != null)
                if (last + 1 != values[r][c])
                    return false;
                last++;
    return true;

```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

(b) Write the `SlidingPuzzle` method `initialize`. Method `initialize` fills the `side × side` values `grid` with random integers 0 through  $side^2 - 1$ , without repeating numbers.

Your implementation must use the following algorithm.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through  $n = side^2 - 1$ . The code for this step is provided.

Starting with the first element of the `grid` values, repeat steps 2 and 3 until the `grid` has been filled.

2. Pick a random element from `temp` and place that element into the next empty `grid` location.
3. Remove that element from `temp` by calling the `ArrayList` `remove` method on its index.

Complete method `initialize` below. The method has been started for your convenience.

```
/** Initializes the puzzle by placing numbers 0 through  $side^2 - 1$  into random locations
 */
public void initialize()
{
    ArrayList<Integer> temp = new ArrayList<Integer>();
    for (int j = 0; j < side * side; j++)
        temp.add(new Integer(j));
```

// Write your solution below.

```
temp.add(null) // this is so a random location has the first element
for (int r = 0; r < side; r++) {
    for (int c = 0; c < side; c++) {
        Integer i = temp.remove(Math.random() * temp.size());
        values[r][c] = i;
    }
}
```

```
}
```

**GO ON TO THE NEXT PAGE.**

- (c) What is the expected big-Oh running time of the initialization algorithm described in part (b), in terms of  $n$ , the number of tiles?

$$O(n^2)$$

where  $n$  is the amount of jobs  
 This is because the array list has to shift everything down

- (d) Consider a variation of the algorithm described in part (b) in which Step 3 is changed.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through  $n = \text{side}^2 - 1$ .

Starting with the first element of the grid values, repeat steps 2 and 3 until the grid has been filled.

2. Pick a random element from `temp` and place that element into the next empty grid location.
3. Replace that element with the element in the last index of `temp`, then remove the last element from `temp`. (Note: removing the last item of an `ArrayList` is a constant time operation.)

What is the expected big-Oh running time of this variation in terms of  $n$ , the number of tiles?

$$O(n)$$

The array list does not have to shift its elements down

**GO ON TO THE NEXT PAGE.**

ABIC

- (a) Write the SlidingPuzzle method isDone. Method isDone returns true if the values in the puzzle appear in increasing order when traversed in row-major order; otherwise, it returns false. The value 0 (denoting the hole) may appear anywhere within the puzzle.

Complete method isDone below.

```
/** Precondition: the puzzle grid contains the distinct values 0 through side2 - 1
 * @return true if the tiles in the puzzle are all arranged in increasing order
 *         (the hole value 0 may be in any position);
 *         false otherwise
 */
public boolean isDone() {
    int[] arr = new int[side];
    // arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
    for (int i = 0; i < values.length; i++) {
        for (int k = 0; k < values[i].length; k++) {
            if (values[i][k] == arr[i][k])
                continue;
            else if (values[i][k] == 0)
                continue;
            else {
                return false;
            }
        }
    }
    return true;
}
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

AB1c

(b) Write the `SlidingPuzzle` method `initialize`. Method `initialize` fills the `side × side` values `grid` with random integers 0 through  $side^2 - 1$ , without repeating numbers.

Your implementation must use the following algorithm.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through  $n = side^2 - 1$ . The code for this step is provided.

Starting with the first element of the `grid` values, repeat steps 2 and 3 until the grid has been filled.

2. Pick a random element from `temp` and place that element into the next empty grid location.
3. Remove that element from `temp` by calling the `ArrayList` `remove` method on its index.

Complete method `initialize` below. The method has been started for your convenience.

```
/** Initializes the puzzle by placing numbers 0 through  $side^2 - 1$  into random locations
 */
```

```
public void initialize()
```

```
{
```

```
    ArrayList<Integer> temp = new ArrayList<Integer>();
```

```
    for (int j = 0; j < side * side; j++)
```

```
        temp.add(new Integer(j));
```

```
    // Write your solution below.
```

```
        int randVal = 0;
```

```
        for (int i = 0; i < temp.size(); i++) {
```

```
            randVal = Math.random() * (side * side - 1);
```

```
            Integer val = temp.get(i);
```

```
            values[i] = val;
```

```
            temp.remove(i);
```

```
        }
```

```
    }
```

```
}
```

GO ON TO THE NEXT PAGE.

- (c) What is the expected big-Oh running time of the initialization algorithm described in part (b), in terms of  $n$ , the number of tiles?

The big-Oh running time is gonna be  $O(n^2)$  because of the two for loops in that method

- (d) Consider a variation of the algorithm described in part (b) in which Step 3 is changed.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through  $n = \text{side}^2 - 1$ .

Starting with the first element of the grid values, repeat steps 2 and 3 until the grid has been filled.

2. Pick a random element from `temp` and place that element into the next empty grid location.
3. Replace that element with the element in the last index of `temp`, then remove the last element from `temp`. (Note: removing the last item of an `ArrayList` is a constant time operation.)

What is the expected big-Oh running time of this variation in terms of  $n$ , the number of tiles?

It would probably take longer

**GO ON TO THE NEXT PAGE.**

# AP<sup>®</sup> COMPUTER SCIENCE AB 2007 SCORING COMMENTARY

## Question 1

### Overview

This question focused on two-dimensional array access, algorithm implementation, and analysis. Students were provided a class framework for a sliding puzzle board, with a 2-D array field and constructor. In part (a) they were required to complete the `isDone` method, which checks to see if the numbered tiles are in order in the board. This involved traversing the 2-D array, comparing each entry with the previous one (or with a running counter), and ignoring the space. In part (b) students were given an algorithm for initializing a puzzle board and were required to implement that algorithm. This involved repeatedly selecting an `ArrayList` element at random, copying it to the puzzle board, and then removing that element from the `ArrayList`. In parts (c) and (d) they were asked to identify the big-Oh complexity of the described algorithm and a variation in which one step was replaced.

### Sample: AB1a

#### Score: 8

In this solution the method `isDone` fails to detect the case in which the out-of-order elements appear at the end of one row and the beginning of the following row, respectively. For this reason, the method does not always return `false` when the arrangement of integers in the array does not constitute a solution of the puzzle. Because of this error the student lost the 1 point assigned to “return false if numbers out of order.” The student earned all the other points.

In part (a) the `isDone` method accesses elements of values inside a pair of nested loops. The upper bound on the inner loop is `(side - 1)` rather than `side`, but one of the references to the elements of the array adds 1 to the column index, so the loop examines every element of the array. The `isDone` method returns `true` in all cases in which it does not return `false`. It only returns `false` when elements in a given row are out of order. The method does attempt to skip over the zero in the array. This is enough to earn the 1 point for returning `true` whenever the arrangement of integers in the array does constitute a solution of the puzzle.

In part (b) the `initialize` method correctly generates random integers in the right range, reads the values of elements in `temp` with calls to `get()`, assigns those elements to `values`, and then removes them from `temp`. The method touches every element in the list and in the array.

In parts (c) and (d) the student correctly describes the complexity of the two initialization algorithms by providing the correct big-Oh expressions.

# AP<sup>®</sup> COMPUTER SCIENCE AB 2007 SCORING COMMENTARY

## Question 1 (continued)

### Sample: AB1b

Score: 6

In part (a) the `isDone` method accesses all elements of `values`. The bounds on the loops span the dimensions of the array, and the indices in the references to elements of the array remain within bounds. The student earned 1½ points for the 2-D traversal. The method `isDone` does not allow for the fact that zero may appear anywhere in a solution, so it does not return `true` whenever the arrangement of integers represents a solution of the puzzle. Thus the 1 point for returning true for a correct solution was not awarded. The test against `null` is incorrect. This error prevents the method from recognizing arrangements that are not solutions, so 1 point for returning false if the numbers were out of order was lost.

In part (b) the addition of a `null` element to the `temp` list changes its size. The student computes the allowed range of random values by calling `temp.size()`. That size (and so the range) is incorrect because of the extra element in `temp`. Because of this error the student lost the 1 point assigned to returns a random number in the range `0 ... temp.size`. The student earned all other points in part (b) since the `initialize` method accesses elements in `temp` and, at the same time, removes elements from `temp` with calls to `remove()`. It copies integers from `temp` to `values` without overwriting elements of `values`. The loop transfers every element from `temp` to `values`.

In parts (c) and (d) the student correctly characterizes the complexity of two algorithms for the initialization of the two-dimensional array with random integers.

### Sample: AB1c

Score: 2

In part (a) the student references elements of `values` within loops that scan the whole width and height of that array. In part (c) the student also recognizes that the first initialization algorithm is  $O(n^2)$ . Together these two parts of the response netted the student 2 points. All other points were lost. The method of part (a) does not correctly distinguish between arrangements of integers that solve the puzzle and those that do not. The student has written a test that refers to a second array whose values are fixed. The test cannot work for a puzzle of arbitrary size.

In part (b) the `initialize()` method does not produce random integers in the right range. The method specifies the wrong range and calls the `Math.random()` method incorrectly. The parameter in the call to `get()` is not the generated (random) index. The reference to `values` in the assignment statement has only a single index where two are needed. The parameter in the call to `remove()` is not the random index but the loop counter. The loop does not transfer every element of `temp` to `values`.

Part (d) called for an expression. The sentence that the student has written in place of an expression does not quantify the order of the algorithm's complexity. It incorrectly asserts that execution of the alternative algorithm "would probably take longer" than the execution of the first initialization algorithm.