```
    Integer target = getTarget(array1, array2);
    if (target == null) return null;

    for (int one : array1) {
       for (int two : array2) {
          if (one - two == target) {
             int[] values = {one, two};
             return values;
          }
       }
    }

    return null;
 }

 Integer getTarget(int[] array1, int[] array2) {
    int sum1 = sum(array1);
    int sum2 = sum(array2);

    if ((sum1 - sum2) % 2 != 0) return null;
    return (sum1 - sum2) / 2;
 }
```

We've used an `Integer` (a boxed data type) as the return value for `getTarget`. This allows us to distinguish an "error" case.

This algorithm takes O(AB) time.

Optimal Solution

This problem reduces to finding a pair of values that have a particular difference. With that in mind, let's revisit what the brute force does.

In the brute force, we're looping through A and then, for each element, looking for an element in B which gives us the "right" difference. If the value in A is 5 and the target is 3, then we must be looking for the value 2. That's the only value that could fulfill the goal.

That is, rather than writing `one - two == target`, we could have written `two == one - target`. How can we more quickly find an element in B that equals `one - target`?

We can do this very quickly with a hash table. We just throw all the elements in B into a hash table. Then, iterate through A and look for the appropriate element in B.

```
1  int[] findSwapValues(int[] array1, int[] array2) {
2     Integer target = getTarget(array1, array2);
3     if (target == null) return null;
4     return findDifference(array1, array2, target);
5  }
6
7  /* Find a pair of values with a specific difference. */
8  int[] findDifference(int[] array1, int[] array2, int target) {
9     HashSet<Integer> contents2 = getContents(array2);
10    for (int one : array1) {
11       int two = one - target;
12       if (contents2.contains(two)) {
13          int[] values = {one, two};
14          return values;
15       }
```

```
  }

  return null;
}

/* Put contents of array into hash set. */
HashSet<Integer> getContents(int[] array) {
  HashSet<Integer> set = new HashSet<Integer>();
  for (int a : array) {
    set.add(a);
  }
  return set;
}
```

This solution will take O(A+B) time. This is the Best Conceivable Runtime (BCR), since we have to at least touch every element in the two arrays.

Alternate Solution

If the arrays are sorted, we can iterate through them to find an appropriate pair. This will require less space.

```
int[] findSwapValues(int[] array1, int[] array2) {
  Integer target = getTarget(array1, array2);
  if (target == null) return null;
  return findDifference(array1, array2, target);
}

int[] findDifference(int[] array1, int[] array2, int target) {
  int a = 0;
  int b = 0;

  while (a < array1.length && b < array2.length) {
    int difference = array1[a] - array2[b];
    /* Compare difference to target. If difference is too small, then make it
     * bigger by moving a to a bigger value. If it is too big, then make it
     * smaller by moving b to a bigger value. If it's just right, return this
     * pair. */
    if (difference == target) {
      int[] values = {array1[a], array2[b]};
      return values;
    } else if (difference < target) {
      a++;
    } else {
      b++;
    }
  }

  return null;
}
```

This algorithm takes O(A + B) time but requires the arrays to be sorted. If the arrays aren't sorted, we can still apply this algorithm but we'd have to sort the arrays first. The overall runtime would be O(A log A + B log B).

16.22 Langton's Ant: An ant is sitting on an infinite grid of white and black squares. It initially faces right. At each step, it does the following:

(1) At a white square, flip the color of the square, turn 90 degrees right (clockwise), and move forward one unit.

(2) At a black square, flip the color of the square, turn 90 degrees left (counter-clockwise), and move forward one unit.

Write a program to simulate the first K moves that the ant makes and print the final board as a grid. Note that you are not provided with the data structure to represent the grid. This is something you must design yourself. The only input to your method is K. You should print the final grid and return nothing. The method signature might be something like `void printKMoves(int K)`.

pg 185

SOLUTION

At first glance, this problem seems very straightforward: create a grid, remember the ant's position and orientation, flip the cells, turn, and move. The interesting part comes in how to handle an infinite grid.

Solution #1: Fixed Array

Technically, since we're only running the first K moves, we do have a max size for the grid. The ant cannot move more than K moves in either direction. If we create a grid that has width 2K and height 2K (and place the ant at the center), we know it will be big enough.

The problem with this is that it's not very extensible. If you run K moves and then want to run another K moves, you might be out of luck.

Additionally, this solution wastes a good amount of space. The max might be K moves in a particular dimension, but the ant is probably going in circles a bit. You probably won't need all this space.

Solution #2: Resizable Array

One thought is to use a resizable array, such as Java's `ArrayList` class. This allows us to grow an array as necessary, while still offering `O(1)` amortized insertion.

The problem is that our grid needs to grow in two dimensions, but the `ArrayList` is only a single array. Additionally, we need to grow "backward" into negative values. The `ArrayList` class doesn't support this.

However, we take a similar approach by building our own resizable grid. Each time the ant hits an edge, we double the size of the grid in that dimension.

What about the negative expansions? While conceptually we can talk about something being at negative positions, we cannot actually access array indices with negative values.

One way we can handle this is to create "fake indices." Let us treat the ant as being at coordinates `(-3, -10)`, but track some sort of offset or delta to translate these coordinates into array indices.

This is actually unnecessary, though. The ant's location does not need to be publicly exposed or consistent (unless, of course, indicated by the interviewer). When the ant travels into negative coordinates, we can double the size of the array and just move the ant and all cells into the positive coordinates. Essentially, we are relabeling all the indices.

This relabeling will not impact the big O time since we have to create a new matrix anyway.

```
1  public class Grid {
2     private boolean[][] grid;
```

```
3     private Ant ant = new Ant();
4
5     public Grid() {
6        grid = new boolean[1][1];
7     }
8
9     /* Copy old values into new array, with an offset/shift applied to the row and
10     * columns. * /
11    private void copyWithShift(boolean[][] oldGrid, boolean[][] newGrid,
12                               int shiftRow, int shiftColumn) {
13       for (int r = 0; r < oldGrid.length; r++) {
14          for (int c = 0; c < oldGrid[0].length; c++) {
15             newGrid[r + shiftRow][c + shiftColumn] = oldGrid[r][c];
16          }
17       }
18    }
19
20    /* Ensure that the given position will fit on the array. If necessary, double
21     * the size of the matrix, copy the old values over, and adjust the ant's
22     * position so that it's in a positive range. * /
23    private void ensureFit(Position position) {
24       int shiftRow = 0;
25       int shiftColumn = 0;
26
27       /* Calculate new number of rows. * /
28       int numRows = grid.length;
29       if (position.row < 0) {
30          shiftRow = numRows;
31          numRows * =2;
32       } else if (position.row >= numRows) {
33          numRows * =2;
34       }
35
36       /* Calculate new number of columns. * /
37       int numColumns = grid[0].length;
38       if (position.column < 0) {
39          shiftColumn = numColumns;
40          numColumns * =2;
41       } else if (position.column >= numColumns) {
42          numColumns * =2;
43       }
44
45       /* Grow array, if necessary. Shift ant's position too. * /
46       if (numRows != grid.length || numColumns != grid[0].length) {
47          boolean[][] newGrid = new boolean[numRows][numColumns];
48          copyWithShift(grid, newGrid, shiftRow, shiftColumn);
49          ant.adjustPosition(shiftRow, shiftColumn);
50          grid = newGrid;
51       }
52    }
53
54    /* Flip color of cells. * /
55    private void flip(Position position) {
56       int row = position.row;
57       int column = position.column;
58       grid[row][column] = grid[row][column] ? false : true;
```

```
    }

    /* Move ant. */
    public void move() {
        ant.turn(grid[ant.position.row][ant.position.column]);
        flip(ant.position);
        ant.move();
        ensureFit(ant.position); // grow
    }

    /* Print board. */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (r == ant.position.row && c == ant.position.column) {
                    sb.append(ant.orientation);
                } else if (grid[r][c]) {
                    sb.append("X");
                } else {
                    sb.append("_");
                }
            }
            sb.append("\n");
        }
        sb.append("Ant: " + ant.orientation + ". \n");
        return sb.toString();
    }
}
```

We pulled the Ant code into a separate class. The nice thing about this is that if we need to have multiple ants for some reason, we can easily extend the code to support this.

```
public class Ant {
    public Position position = new Position(0, 0);
    public Orientation orientation = Orientation.right;

    public void turn(boolean clockwise) {
        orientation = orientation.getTurn(clockwise);
    }

    public void move() {
        if (orientation == Orientation.left) {
            position.column--;
        } else if (orientation == Orientation.right) {
            position.column++;
        } else if (orientation == Orientation.up) {
            position.row--;
        } else if (orientation == Orientation.down) {
            position.row++;
        }
    }

    public void adjustPosition(int shiftRow, int shiftColumn) {
        position.row += shiftRow;
        position.column += shiftColumn;
    }
}
```

Orientation is also its own enum, with a few useful functions.

```
public enum Orientation {
   left, up, right, down;

   public Orientation getTurn(boolean clockwise) {
      if (this == left) {
         return clockwise ? up : down;
      } else if (this == up) {
         return clockwise ? right : left;
      } else if (this == right) {
         return clockwise ? down : up;
      } else { // down
         return clockwise ? left : right;
      }
   }

   @Override
   public String toString() {
      if (this == left) {
         return "\u2190";
      } else if (this == up) {
         return "\u2191";
      } else if (this == right) {
         return "\u2192";
      } else { // down
         return "\u2193";
      }
   }
}
```

We've also put Position into its own simple class. We could just as easily track the row and column separately.

```
public class Position {
   public int row;
   public int column;

   public Position(int row, int column) {
      this.row = row;
      this.column = column;
   }
}
```

This works, but it's actually more complicated than is necessary.

Solution #3: HashSet

Although it may seem "obvious" that we would use a matrix to represent a grid, it's actually easier not to do that. All we actually need is a list of the white squares (as well as the ant's location and orientation).

We can do this by using a HashSet of the white squares. If a position is in the hash set, then the square is white. Otherwise, it is black.

The one tricky bit is how to print the board. Where do we start printing? Where do we end?

Since we will need to print a grid, we can track what should be top-left and bottom-right corner of the grid. Each time the ant moves, we compare the ant's position to the most top-left position and most bottom-right position, updating them if necessary.

```
public class Board {
   private HashSet<Position> whites = new HashSet<Position>();
   private Ant ant = new Ant();
   private Position topLeftCorner = new Position(0, 0);
   private Position bottomRightCorner = new Position(0, 0);

   public Board() { }

   /* Move ant. */
   public void move() {
      ant.turn(isWhite(ant.position)); // Turn
      flip(ant.position); // flip
      ant.move(); // move
      ensureFit(ant.position);
   }

   /* Flip color of cells. */
   private void flip(Position position) {
      if (whites.contains(position)) {
         whites.remove(position);
      } else {
         whites.add(position.clone());
      }
   }

   /* Grow grid by tracking the most top-left and bottom-right positions.*/
   private void ensureFit(Position position) {
      int row = position.row;
      int column = position.column;

      topLeftCorner.row = Math.min(topLeftCorner.row, row);
      topLeftCorner.column = Math.min(topLeftCorner.column, column);

      bottomRightCorner.row = Math.max(bottomRightCorner.row, row);
      bottomRightCorner.column = Math.max(bottomRightCorner.column, column);
   }

   /* Check if cell is white. */
   public boolean isWhite(Position p) {
      return whites.contains(p);
   }

   /* Check if cell is white. */
   public boolean isWhite(int row, int column) {
      return whites.contains(new Position(row, column));
   }

   /* Print board. */
   public String toString() {
      StringBuilder sb = new StringBuilder();
      int rowMin = topLeftCorner.row;
      int rowMax = bottomRightCorner.row;
      int colMin = topLeftCorner.column;
      int colMax = bottomRightCorner.column;
      for (int r = rowMin; r <= rowMax; r++) {
         for (int c = colMin; c <= colMax; c++) {
```

```
57              if (r == ant.position.row && c == ant.position.column) {
58                 sb.append(ant.orientation);
59              } else if (isWhite(r, c)) {
60                 sb.append("X");
61              } else {
62                 sb.append("_");
63              }
64           }
65           sb.append("\n");
66        }
67        sb.append("Ant: " + ant.orientation + ". \n");
68        return sb.toString();
69     }
```

The implementation of `Ant` and `Orientation` is the same.

The implementation of `Position` gets updated slightly, in order to support the `HashSet` functionality. The position will be the key, so we need to implement a `hashCode()` function.

```
1  public class Position {
2     public int row;
3     public int column;
4
5     public Position(int row, int column) {
6        this.row = row;
7        this.column = column;
8     }
9
10    @Override
11    public boolean equals(Object o) {
12       if (o instanceof Position) {
13          Position p = (Position) o;
14          return p.row == row && p.column == column;
15       }
16       return false;
17    }
18
19    @Override
20    public int hashCode() {
21       /* There are many options for hash functions. This is one. */
22       return (row * 31) ^ column;
23    }
24
25    public Position clone() {
26       return new Position(row, column);
27    }
28 }
```

The nice thing about this implementation is that if we do need to access a particular cell elsewhere, we have consistent row and column labeling.

16.23 Rand7 from Rand5: Implement a method `rand7()` given `rand5()`. That is, given a method that generates a random number between 0 and 4 (inclusive), write a method that generates a random number between 0 and 6 (inclusive).

pg 186

SOLUTION

To implement this function correctly, we must have each of the values between 0 and 6 returned with 1/7th probability.

First Attempt (Fixed Number of Calls)

As a first attempt, we might try generating all numbers between 0 and 9, and then mod the resulting value by 7. Our code for it might look something like this:

```
1  int rand7() {
2    int v = rand5() + rand5();
3    return v % 7;
4  }
```

Unfortunately, the above code will not generate the values with equal probability. We can see this by looking at the results of each call to `rand5()` an d the return result of the `rand7()` function.

1st Call	2nd Call	Result
0	0	0
0	1	1
0	2	2
0	3	3
0	4	4
1	0	1
1	1	2
1	2	3
1	3	4
1	4	5
2	0	2
2	1	3
2	2	4

1st Call	2nd Call	Result
2	3	5
2	4	6
3	0	3
3	1	4
3	2	5
3	3	6
3	4	0
4	0	4
4	1	5
4	2	6
4	3	0
4	4	1

Each individual row has a 1 in 25 chance of occurring, since there are two calls to `rand5()` and each distributes its results with $1/5$ th probability. If you count up the number of times each number occurs, you'll note that this `rand7()` function will return 4 with $5/25$ th probability but return 0 with just $3/25$ th probability. This means that our function has failed; the results do not have probability $1/7$ th.

Now, imagine we modify our function to add an if-statement, to change the constant multiplier, or to insert a new call to `rand5()`. We will still wind up with a similar looking table, and the probability of getting any one of those rows will be $1/5^k$, where k is the number of calls to `rand5()` in that row. Different rows may have different number of calls.

The probability of winding up with the result of the `rand7()` function being, say, 6 would be the sum of the probabilities of all rows that result in 6. That is:

$$P(\text{rand7}() = 6) = 1/5^i + 1/5^j + \ldots + 1/5^m$$

We know that, in order for our function to be correct, this probability must equal $1/7$. This is impossible though. Because 5 and 7 are relatively prime, no series of reciprocal powers of 5 will result in $1/7$.

Does this mean the problem is impossible? Not exactly. Strictly speaking, it means that, as long as we can list out the combinations of `rand5()` results that will result in a particular value of `rand7()`, the function will not give well distributed results.

We can still solve this problem. We just have to use a while loop, and realize that there's no telling just how many turns will be required to return a result.

Second Attempt (Nondeterministic Number of Calls)

As soon as we've allowed for a while loop, our work gets much easier. We just need to generate a range of values where each value is equally likely (and where the range has at least seven elements). If we can do this, then we can discard the elements greater than the previous multiple of 7, and mod the rest of them by 7. This will get us a value within the range of 0 to 6, with each value being equally likely.

In the below code, we generate the range 0 through 24 by doing `5 * rand5() + rand5()`. Then, we discard the values between 21 and 24, since they would otherwise make `rand7()` unfairly weighted towards 0 through 3. Finally, we mod by 7 to give us the values in the range 0 to 6 with equal probability.

Note that because we discard values in this approach, we have no guarantee on the number of `rand5()` calls it may take to return a value. This is what is meant by a *nondeterministic* number of calls.

```
1  int rand7() {
2    while (true) {
3      int num = 5 * rand5() + rand5();
4      if (num < 21) {
5        return num % 7;
6      }
7    }
8  }
```

Observe that doing `5 * rand5() + rand5()` gives us exactly one way of getting each number in its range (0 to 24). This ensures that each value is equally probable.

Could we instead do `2 * rand5() + rand5()`? No, because the values wouldn't be equally distributed. For example, there would be three ways of getting a 6 (`6 = 2 * 1 + 4`, `6 = 2 * 2 + 2`, and `6 = 2 * 3 + 0`) but only one way of getting a 0 (`0=2*0+0`). The values in the range are not equally probable.

There *is* a way that we can use `2 * rand5()` and still get an identically distributed range, but it's much more complicated. See below.

```
1  int rand7() {
2    while (true) {
3      int r1 = 2 * rand5(); / *evens between 0 and 9 */
4      int r2 = rand5(); / *used later to generate a 0 or 1 */
5      if (r2 != 4) { / *r2 has extra even num-discard the extra */
6        int rand1 = r2 % 2; / *Generate 0 or 1 */
7        int num = r1 + rand1; / *will be in the range 0 to 9 */
8        if (num < 7) {
9          return num;
10       }
11     }
12   }
13 }
```