85 can't be in the black area, since 95 is in the upper left hand corner and is therefore the smallest element in that square.

85 can't be in the light gray area either, since 35 is in the lower right hand corner of that square.

85 must be in one of the two white areas.

So, we partition our grid into four quadrants and recursively search the lower left quadrant and the upper right quadrant. These, too, will get divided into quadrants and searched.

Observe that since the diagonal is sorted, we can efficiently search it using binary search.

The code below implements this algorithm.

```
1  Coordinate findElement(int[][] matrix, Coordinate origin, Coordinate dest, int x){
2    if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
3      return null;
4    }
5    if (matrix[origin.row][origin.column] == x) {
6      return origin;
7    } else if (!origin.isBefore(dest)) {
8      return null;
9    }
10
11   /* Set start to start of diagonal and end to the end of the diagonal. Since the
12    * grid may not be square, the end of the diagonal may not equal dest. */
13   Coordinate start = (Coordinate) origin.clone();
14   int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);
15   Coordinate end = new Coordinate(start.row + diagDist, start.column + diagDist);
16   Coordinate p = new Coordinate(0, 0);
17
18   /* Do binary search on the diagonal, looking for the first element > x */
19   while (start.isBefore(end)) {
20     p.setToAverage(start, end);
21     if (x > matrix[p.row][p.column]) {
22       start.row = p.row + 1;
23       start.column = p.column + 1;
24     } else {
25       end.row = p.row - 1;
26       end.column = p.column - 1;
27     }
28   }
29
30   /* Split the grid into quadrants. Search the bottom left and the top right. */
31   return partitionAndSearch(matrix, origin, dest, start, x);
32 }
33
34 Coordinate partitionAndSearch(int[][] matrix, Coordinate origin, Coordinate dest,
35                               Coordinate pivot, int x) {
36   Coordinate lowerLeftOrigin = new Coordinate(pivot.row, origin.column);
37   Coordinate lowerLeftDest = new Coordinate(dest.row, pivot.column - 1);
38   Coordinate upperRightOrigin = new Coordinate(origin.row, pivot.column);
39   Coordinate upperRightDest = new Coordinate(pivot.row - 1, dest.column);
40
41   Coordinate lowerLeft = findElement(matrix, lowerLeftOrigin, lowerLeftDest, x);
42   if (lowerLeft == null) {
43     return findElement(matrix, upperRightOrigin, upperRightDest, x);
44   }
```

```
45    return lowerLeft;
46  }
47
48  Coordinate findElement(int[][] matrix, int x) {
49    Coordinate origin = new Coordinate(0, 0);
50    Coordinate dest = new Coordinate(matrix.length - 1, matrix[0].length - 1);
51    return findElement(matrix, origin, dest, x);
52  }
53
54  public class Coordinate implements Cloneable {
55    public int row, column;
56    public Coordinate(int r, int c) {
57      row = r;
58      column = c;
59    }
60
61    public boolean inbounds(int[][] matrix) {
62      return  row >= 0 && column >= 0 &&
63              row < matrix.length && column < matrix[0].length;
64    }
65
66    public boolean isBefore(Coordinate p) {
67      return row <= p.row && column <= p.column;
68    }
69
70    public Object clone() {
71      return new Coordinate(row, column);
72    }
73
74    public void setToAverage(Coordinate min, Coordinate max) {
75      row = (min.row + max.row) / 2;
76      column = (min.column + max.column) / 2;
77    }
78  }
```

If you read all this code and thought, "there's no way I could do all this in an interview!" you're probably right. You couldn't. But, your performance on any problem is evaluated compared to other candidates on the same problem. So while you couldn't implement all this, neither could they. You are at no disadvantage when you get a tricky problem like this.

You help yourself out a bit by separating code out into other methods. For example, by pulling `partitionAndSearch` out into its own method, you will have an easier time outlining key aspects of the code. You can then come back to fill in the body for `partitionAndSearch` if you have time.

10.10 Rank from Stream: Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

`getRankOfNumber(1) = 0`

`getRankOfNumber(3) = 1`

`getRankOfNumber(4) = 3`

pg 151

SOLUTION

A relatively easy way to implement this would be to have an array that holds all the elements in sorted order. When a new element comes in, we would need to shift the other elements to make room. Implementing `getRankOfNumber` would be quite efficient, though. We would simply perform a binary search for n, and return the index.

However, this is very inefficient for inserting elements (that is, the `track(int x)` function). We need a data structure which is good at keeping relative ordering, as well as updating when we insert new elements. A binary search tree can do just that.

Instead of inserting elements into an array, we insert elements into a binary search tree. The method `track(int x)` will run in `O(log n)` time, where n is the size of the tree (provided, of course, that the tree is balanced).

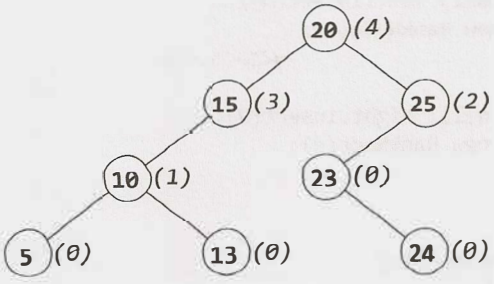
To find the rank of a number, we could do an in-order traversal, keeping a counter as we traverse. The goal is that, by the time we find x, `counter` will equal the number of elements less than x.

As long as we're moving left during searching for x, the counter won't change. Why? Because all the values we're skipping on the right side are greater than x. After all, the very smallest element (with rank of 1) is the leftmost node.

When we move to the right though, we skip over a bunch of elements on the left. All of these elements are less than x, so we'll need to increment `counter` by the number of elements in the left subtree.

Rather than counting the size of the left subtree (which would be inefficient), we can track this information as we add new elements to the tree.

Let's walk through an example on the following tree. In the below example, the value in parentheses indicates the number of nodes in the left subtree (or, in other words, the rank of the node *relative* to its subtree).



Suppose we want to find the rank of 24 in the tree above. We would compare 24 with the root, 20, and find that 24 must reside on the right. The root has 4 nodes in its left subtree, and when we include the root itself, this gives us five total nodes smaller than 24. We set `counter` to 5.

Then, we compare 24 with node 25 and find that 24 must be on the left. The value of `counter` does not update, since we're not "passing over" any smaller nodes. The value of `counter` is still 5.

Next, we compare 24 with node 23, and find that 24 must be on the right. `Counter` gets incremented by just 1 (to 6), since 23 has no left nodes.

Finally, we find 24 and we return `counter`: 6.

Recursively, the algorithm is the following:

```
int getRank(Node node, int x) {
  if x is node.data, return node.leftSize()
  if x is on left of node, return getRank(node.left, x)
  if x is on right of node, return node.leftSize() + 1 + getRank(node.right, x)
}
```

The full code for this is below.

```
RankNode root = null;

void track(int number) {
  if (root == null) {
    root = new RankNode(number);
  } else {
    root.insert(number);
  }
}

int getRankOfNumber(int number) {
  return root.getRank(number);
}


public class RankNode {
  public int left_size = 0;
  public RankNode left, right;
  public int data = 0;
  public RankNode(int d) {
    data = d;
  }

  public void insert(int d) {
    if (d <= data) {
```

```
26          if (left != null) left.insert(d);
27          else left = new RankNode(d);
28          left_size++;
29        } else {
30          if (right != null) right.insert(d);
31          else right = new RankNode(d);
32        }
33      }
34
35      public int getRank(int d) {
36        if (d == data) {
37          return left_size;
38        } else if (d < data) {
39          if (left == null) return -1;
40          else return left.getRank(d);
41        } else {
42          int right_rank = right == null ? -1 : right.getRank(d);
43          if (right_rank == -1) return -1;
44          else return left_size + 1 + right_rank;
45        }
46      }
47  }
```

The `track` method and the `getRankOfNumber` method will both operate in `O(log N)` on a balanced tree and `O(N)` on an unbalanced tree.

Note how we've handled the case in which d is not found in the tree. We check for the -1 return value, and, when we find it, return -1 up the tree. It is important that you handle cases like this.

10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

pg 151

SOLUTION

Since this problem asks us to sort the array in a particular way, one thing we can try is doing a normal sort and then "fixing" the array into an alternating sequence of peaks and valleys.

Suboptimal Solution

Imagine we were given an unsorted array and then sort it to become the following:

```
0  1  4  7  8  9
```

We now have an ascending list of integers.

How can we rearrange this into a proper alternating sequence of peaks and valleys? Let's walk through it and try to do that.

- The `0` is okay.

- The 1 is in the wrong place. We can swap it with either the 0 or 4. Let's swap it with the 0.

  ```
  1  0  4  7  8  9
  ```

- The 4 is okay.
- The 7 is in the wrong place. We can swap it with either the 4 or the 8. Let's swap it with the 4.

  ```
  1  0  7  4  8  9
  ```

- The 9 is in the wrong place. Let's swap it with the 8.

  ```
  1  0  7  4  9  8
  ```

Observe that there's nothing special about the array having these values. The relative order of the elements matters, but all sorted arrays will have the same relative order. Therefore, we can take this same approach on any sorted array.

Before coding, we should clarify the exact algorithm, though.

1. Sort the array in ascending order.
2. Iterate through the elements, starting from index 1 (not 0) and jumping two elements at a time.
3. At each element, swap it with the previous element. Since every three elements appear in the order `small <= medium <= large`, swapping these elements will always put medium as a peak: `medium <= small <= large`.

This approach will ensure that the peaks are in the right place: indexes 1, 3, 5, and so on. As long as the odd-numbered elements (the peaks) are bigger than the adjacent elements, then the even-numbered elements (the valleys) must be smaller than the adjacent elements.

The code to implement this is below.

```
1  void sortValleyPeak(int[] array) {
2    Arrays.sort(array);
3    for (int i = 1; i < array.length; i += 2) {
4      swap(array, i - 1, i);
5    }
6  }
7
8  void swap(int[] array, int left, int right) {
9    int temp = array[left];
10   array[left] = array[right];
11   array[right] = temp;
12 }
```

This algorithm runs in `O(n log n)` time.

Optimal Solution

To optimize past the prior solution, we need to cut out the sorting step. The algorithm must operate on an unsorted array.

Let's revisit an example.

```
9  1  0  4  8  7
```

For each element, we'll look at the adjacent elements. Let's imagine some sequences. We'll just use the numbers 0, 1 and 2. The specific values don't matter.

```
0  1  2
0  2  1       // peak
1  0  2
1  2  0       // peak
2  1  0
```

```
2   0   1
```

If the center element needs to be a peak, then two of those sequences work. Can we fix the other ones to make the center element a peak?

Yes. We can fix this sequence by swapping the center element with the largest adjacent element.

```
0   1   2   ->  0   2   1
0   2   1       // peak
1   0   2   ->  1   2   0
1   2   0   // peak
2   1   0   ->  1   2   0
2   0   1   ->  0   2   1
```

As we noted before, if we make sure the peaks are in the right place then we know the valleys are in the right place.

> We should be a little cautious here. Is it possible that one of these swaps could "break" an earlier part of the sequence that we'd already processed? This is a good thing to worry about, but it's not an issue here. If we're swapping `middle` with `left`, then `left` is currently a valley. `Middle` is smaller than `left`, so we're putting an even smaller element as a valley. Nothing will break. All is good!

The code to implement this is below.

```
1   void sortValleyPeak(int[] array) {
2     for (int i = 1; i < array.length; i += 2) {
3       int biggestIndex = maxIndex(array, i - 1, i, i + 1);
4       if (i != biggestIndex) {
5         swap(array, i, biggestIndex);
6       }
7     }
8   }
9
10  int maxIndex(int[] array, int a, int b, int c) {
11    int len = array.length;
12    int aValue = a >= 0 && a < len ? array[a] : Integer.MIN_VALUE;
13    int bValue = b >= 0 && b < len ? array[b] : Integer.MIN_VALUE;
14    int cValue = c >= 0 && c < len ? array[c] : Integer.MIN_VALUE;
15
16    int max = Math.max(aValue, Math.max(bValue, cValue));
17    if (aValue == max) return a;
18    else if (bValue == max) return b;
19    else return c;
20  }
```

This algorithm takes O(n) time.

11

Solutions to Testing

11.1 **Mistake:** Find the mistake(s) in the following code:

```
unsigned int i;
for (i = 100; i >= 0; --i)
  printf("%d\n", i);
```

pg 157

SOLUTION

There are two mistakes in this code.

First, note that an `unsigned int` is, by definition, always greater than or equal to zero. The for loop condition will therefore always be true, and it will loop infinitely.

The correct code to print all numbers from 100 to 1, is `i > 0`. If we truly wanted to print zero, we could add an additional `printf` statement after the for loop.

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%d\n", i);
```

One additional correction is to use %u in place of %d, as we are printing `unsigned int`.

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%u\n", i);
```

This code will now correctly print the list of all numbers from 100 to 1, in descending order.

11.2 **Random Crashes:** You are given the source to an application which crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is single threaded, and uses only the C standard library. What programming errors could be causing this crash? How would you test each one?

pg 157

SOLUTION

The question largely depends on the type of application being diagnosed. However, we can give some general causes of random crashes.

1. *"Random Variable:"* The application may use some random number or variable component that may not be fixed for every execution of the program. Examples include user input, a random number generated by the program, or the time of day.

2. *Uninitialized Variable:* The application could have an uninitialized variable which, in some languages, may cause it to take on an arbitrary value. The values of this variable could result in the code taking a slightly different path each time.
3. *Memory Leak:* The program may have run out of memory. Other culprits are totally random for each run since it depends on the number of processes running at that particular time. This also includes heap overflow or corruption of data on the stack.
4. *External Dependencies:* The program may depend on another application, machine, or resource. If there are multiple dependencies, the program could crash at any point.

To track down the issue, we should start with learning as much as possible about the application. Who is running it? What are they doing with it? What kind of application is it?

Additionally, although the application doesn't crash in exactly the same place, it's possible that it is linked to specific components or scenarios. For example, it could be that the application never crashes if it's simply launched and left untouched, and that crashes only appear at some point after loading a file. Or, it may be that all the crashes take place within the lower level components, such as file I/O.

It may be useful to approach this by elimination. Close down all other applications on the system. Track resource use very carefully. If there are parts of the program we can disable, do so. Run it on a different machine and see if we experience the same issue. The more we can eliminate (or change), the easier we can track down the issue.

Additionally, we may be able to use tools to check for specific situations. For example, to investigate issue #2, we can utilize runtime tools which check for uninitialized variables.

These problems are as much about your brainstorming ability as they are about your approach. Do you jump all over the place, shouting out random suggestions? Or do you approach it in a logical, structured manner? Hopefully, it's the latter.

11.3 Chess Test: We have the following method used in a chess game: boolean `canMoveTo(int x, int y)`. This method is part of the `Piece` class and returns whether or not the piece can move to position `(x, y)`. Explain how you would test this method.

pg 157

SOLUTION

In this problem, there are two primary types of testing: extreme case validation (ensuring that the program doesn't crash on bad input), and general case testing. We'll start with the first type.

Testing Type #1: Extreme Case Validation

We need to ensure that the program handles bad or unusual input gracefully. This means checking the following conditions:

- Test with negative numbers for x and y
- Test with x larger than the width
- Test with y larger than the height
- Test with a completely full board
- Test with an empty or nearly empty board
- Test with far more white pieces than black

- Test with far more black pieces than white

For the error cases above, we should ask our interviewer whether we want to return false or throw an exception, and we should test accordingly.

Testing Type #2: General Testing:

General testing is much more expansive. Ideally, we would test every possible board, but there are far too many boards. We can, however, perform a reasonable coverage of different boards.

There are 6 pieces in chess, so we can test each piece against every other piece, in every possible direction. This would look something like the below code:

```
1  foreach piece a:
2    for each other type of piece b (6 types + empty space)
3      foreach direction d
4        Create a board with piece a.
5        Place piece b in direction d.
6        Try to move - check return value.
```

The key to this problem is recognizing that we can't test every possible scenario, even if we would like to. So, instead, we must focus on the essential areas.

11.4 No Test Tools: How would you load test a webpage without using any test tools?

pg 157

SOLUTION

Load testing helps to identify a web application's maximum operating capacity, as well as any bottlenecks that may interfere with its performance. Similarly, it can check how an application responds to variations in load.

To perform load testing, we must first identify the performance critical scenarios and the metrics which fulfill our performance objectives. Typical criteria include:

- Response time
- Throughput
- Resource utilization
- Maximum load that the system can bear.

Then, we design tests to simulate the load, taking care to measure each of these criteria.

In the absence of formal testing tools, we can basically create our own. For example, we could simulate concurrent users by creating thousands of virtual users. We would write a multi-threaded program with thousands of threads, where each thread acts as a real-world user loading the page. For each user, we would programmatically measure response time, data I/O, etc.

We would then analyze the results based on the data gathered during the tests and compare it with the accepted values.