```
          break;
        } else if (j == size - 1) { // Last element
          return first;
        }
      }
    }

    /* Check diagonals. */
    first = board[0][0];
    if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
        if (board[i][i] != first) {
          break;
        } else if (i == size - 1) { // Last element
          return first;
        }
      }
    }

    first = board[0][size - 1];
    if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
        if (board[i][size - i - 1] != first) {
          break;
        } else if (i == size - 1) { // Last element
          return first;
        }
      }
    }

    return Piece.Empty;
  }
```

This is, to the say the least, pretty ugly. We're doing nearly the same work each time. We should look for a way of reusing the code.

Increment and Decrement Function

One way that we can reuse the code better is to just pass in the values to another function that increments/decrements the rows and columns. The hasWon function now just needs the starting position and the amount to increment the row and column by.

```
class Check {
  public int row, column;
  private int rowIncrement, columnIncrement;
  public Check(int row, int column, int rowI, int colI) {
    this.row = row;
    this.column = column;
    this.rowIncrement = rowI;
    this.columnIncrement = colI;
  }

  public void increment() {
    row += rowIncrement;
    column += columnIncrement;
  }

  public boolean inBounds(int size) {
      return row >= 0 && column >= 0 && row < size && column < size;
    }
}

Piece hasWon(Piece[][] board) {
  if (board.length != board[0].length) return Piece.Empty;
  int size = board.length;

  /* Create list of things to check. */
  ArrayList<Check> instructions = new ArrayList<Check>();
  for (int i = 0; i < board.length; i++) {
    instructions.add(new Check(0, i, 1, 0));
    instructions.add(new Check(i, 0, 0, 1));
  }
  instructions.add(new Check(0, 0, 1, 1));
  instructions.add(new Check(0, size - 1, 1, -1));

  /* Check them. */
  for (Check instr : instructions) {
    Piece winner = hasWon(board, instr);
    if (winner != Piece.Empty) {
        return winner;
    }
  }
  return Piece.Empty;
}

Piece hasWon(Piece[][] board, Check instr) {
  Piece first = board[instr.row][instr.column];
  while (instr.inBounds(board.length)) {
    if (board[instr.row][instr.column] != first) {
      return Piece.Empty;
    }
    instr.increment();
  }
  return first;
}
```

The Check function is essentially operating as an iterator.

Iterator

Another way of doing it is, of course, to actually build an iterator.

```
Piece hasWon(Piece[][] board) {
  if (board.length != board[0].length) return Piece.Empty;
  int size = board.length;

  ArrayList<PositionIterator> instructions = new ArrayList<PositionIterator>();
  for (int i = 0; i < board.length; i++) {
    instructions.add(new PositionIterator(new Position(0, i), 1, 0, size));
    instructions.add(new PositionIterator(new Position(i, 0), 0, 1, size));
  }
  instructions.add(new PositionIterator(new Position(0, 0), 1, 1, size));
  instructions.add(new PositionIterator(new Position(0, size - 1), 1, -1, size));

  for (PositionIterator iterator : instructions) {
    Piece winner = hasWon(board, iterator);
    if (winner != Piece.Empty) {
      return winner;
    }
  }
  return Piece.Empty;
}

Piece hasWon(Piece[][] board, PositionIterator iterator) {
  Position firstPosition = iterator.next();
  Piece first = board[firstPosition.row][firstPosition.column];
  while (iterator.hasNext()) {
    Position position = iterator.next();
    if (board[position.row][position.column] != first) {
      return Piece.Empty;
    }
  }
  return first;
}

class PositionIterator implements Iterator<Position> {
  private int rowIncrement, colIncrement, size;
  private Position current;

  public PositionIterator(Position p, int rowIncrement,
                          int colIncrement, int size) {
    this.rowIncrement = rowIncrement;
    this.colIncrement = colIncrement;
    this.size = size;
    current = new Position(p.row - rowIncrement, p.column - colIncrement);
  }

  @Override
  public boolean hasNext() {
    return current.row + rowIncrement < size &&
           current.column + colIncrement < size;
  }

  @Override
  public Position next() {
    current = new Position(current.row + rowIncrement,
                           current.column + colIncrement);
    return current;
  }
}

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

All of this is potentially overkill, but it's worth discussing the options with your interviewer. The point of this problem is to assess your understanding of how to code in a clean and maintainable way.

16.5 Factorial Zeros: Write an algorithm which computes the number of trailing zeros in n factorial.

pg 181

SOLUTION

A simple approach is to compute the factorial, and then count the number of trailing zeros by continuously dividing by ten. The problem with this though is that the bounds of an `int` would be exceeded very quickly. To avoid this issue, we can look at this problem mathematically.

Consider a factorial like `19!`:

```
19! = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

A trailing zero is created with multiples of 10, and multiples of 10 are created with pairs of 5-multiples and 2-multiples.

For example, in 19!, the following terms create the trailing zeros:

```
19! = 2 * ... * 5 * ... * 10 * ... * 15 * 16 * ...
```

Therefore, to count the number of zeros, we only need to count the pairs of multiples of 5 and 2. There will always be more multiples of 2 than 5, though, so simply counting the number of multiples of 5 is sufficient.

One "gotcha" here is 15 contributes a multiple of 5 (and therefore one trailing zero), while 25 contributes two (because `25 = 5 * 5`).

There are two different ways to write this code.

The first way is to iterate through all the numbers from 2 through n, counting the number of times that 5 goes into each number.

```
/* If the number is a 5 of five, return which power of 5. For example: 5 -> 1,
 * 25-> 2, etc. */
int factorsOf5(int i) {
  int count = 0;
  while (i % 5 == 0) {
    count++;
    i /= 5;
  }
  return count;
}

int countFactZeros(int num) {
  int count = 0;
  for (int i = 2; i <= num; i++) {
    count += factorsOf5(i);
  }
  return count;
}
```

This isn't bad, but we can make it a little more efficient by directly counting the factors of 5. Using this approach, we would first count the number of multiples of 5 between 1 and n (which is $^n/_5$), then the number of multiples of 25 ($^n/_{25}$), then 125, and so on.

To count how many multiples of `m` are in n, we can just divide n by `m`.

```
int countFactZeros(int num) {
  int count = 0;
  if (num < 0) {
    return -1;
  }
```

```
    for (int i = 5; num / i > 0; i *= 5) {
      count += num / i;
    }
    return count;
  }
```

This problem is a bit of a brainteaser, but it can be approached logically (as shown above). By thinking through what exactly will contribute a zero, you can come up with a solution. You should be very clear in your rules upfront so that you can implement it correctly.

16.6 Smallest Difference: Given two arrays of integers, compute the pair of values (one value in each array) with the smallest (non-negative) difference. Return the difference.

EXAMPLE

Input: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Output: 3. That is, the pair (11, 8).

pg 181

SOLUTION

Let's start first with a brute force solution.

Brute Force

The simple brute force way is to just iterate through all pairs, compute the difference, and compare it to the current minimum difference.

```
int findSmallestDifference(int[] array1, int[] array2) {
  if (array1.length == 0 || array2.length == 0) return -1;

  int min = Integer.MAX_VALUE;
  for (int i = 0; i < array1.length; i++) {
    for (int j = 0; j < array2.length; j++) {
      if (Math.abs(array1[i] - array2[j]) < min) {
        min = Math.abs(array1[i] - array2[j]);
      }
    }
  }
  return min;
}
```

One minor optimization we could perform from here is to return immediately if we find a difference of zero, since this is the smallest difference possible. However, depending on the input, this might actually be slower.

This will only be faster if there's a pair with difference zero early in the list of pairs. But to add this optimization, we need to execute an additional line of code each time. There's a tradeoff here; it's faster for some inputs and slower for others. Given that it adds complexity in reading the code, it may be best to leave it out.

With or without this "optimization," the algorithm will take O(AB) time.

Optimal

A more optimal approach is to sort the arrays. Once the arrays are sorted, we can find the minimum difference by iterating through the array.

Consider the following two arrays:

```
A: {1, 2, 11, 15}
B: {4, 12, 19, 23, 127, 235}
```

Try the following approach:

1. Suppose a pointer a points to the beginning of A and a pointer b points to the beginning of B. The current difference between a and b is 3. Store this as the `min`.
2. How can we (potentially) make this difference smaller? Well, the value at b is bigger than the value at a, so moving b will only make the difference larger. Therefore, we want to move a.
3. Now a points to 2 and b (still) points to 4. This difference is 2, so we should update `min`. Move a, since it is smaller.
4. Now a points to 11 and b points to 4. Move b.
5. Now a points to 11 and b points to 12. Update `min` to 1. Move b.

And so on.

```
int findSmallestDifference(int[] array1, int[] array2) {
  Arrays.sort(array1);
  Arrays.sort(array2);
  int a = 0;
  int b = 0;
  int difference = Integer.MAX_VALUE;
  while (a < array1.length && b < array2.length) {
    if (Math.abs(array1[a] - array2[b]) < difference) {
      difference = Math.abs(array1[a] - array2[b]);
    }

    /* Move smaller value. */
    if (array1[a] < array2[b]) {
      a++;
    } else {
      b++;
    }
  }
  return difference;
}
```

This algorithm takes `O(A log A + B log B)` time to sort and `O(A + B)` time to find the minimum difference. Therefore, the overall runtime is `O(A log A + B log B)`.

16.7 Number Max: Write a method that finds the maximum of two numbers. You should not use if-else or any other comparison operator.

pg 181

SOLUTION

A common way of implementing a `max` function is to look at the sign of `a - b`. In this case, we can't use a comparison operator on this sign, but we *can* use multiplication.

Let k equal the sign of `a - b` such that if `a - b >= 0`, then k is 1. Else, `k = 0`. Let q be the inverse of k.

We can then implement the code as follows:

```
/* Flips a 1 to a 0 and a 0 to a 1 */
int flip(int bit) {
    return 1^bit;
  }

  /* Returns 1 if a is positive, and 0 if a is negative */
  int sign(int a) {
    return flip((a >> 31) & 0x1);
  }

  int getMaxNaive(int a, int b) {
    int k = sign(a - b);
    int q = flip(k);
    return a * k + b * q;
  }
```

This code almost works. It fails, unfortunately, when a - b overflows. Suppose, for example, that a is INT_MAX - 2 and b is -15. In this case, a - b will be greater than INT_MAX and will overflow, resulting in a negative value.

We can implement a solution to this problem by using the same approach. Our goal is to maintain the condition where k is 1 when a > b. We will need to use more complex logic to accomplish this.

When does a - b overflow? It will overflow only when a is positive and b is negative, or the other way around. It may be difficult to specially detect the overflow condition, but we *can* detect when a and b have different signs. Note that if a and b have different signs, then we want k to equal sign(a).

The logic looks like:

```
if a and b have different signs:
  // if a > 0, then b < 0, and k = 1.
  // if a < 0, then b > 0, and k = 0.
  // so either way, k = sign(a)
  let k = sign(a)
else
  let k = sign(a - b) // overflow is impossible
```

The code below implements this, using multiplication instead of if-statements.

```
int getMax(int a, int b) {
  int c = a - b;

  int sa = sign(a); // if a >= 0, then 1 else 0
  int sb = sign(b); // if b >= 0, then 1 else 0
  int sc = sign(c); // depends on whether or not a - b overflows

  /* Goal: define a value k which is 1 if a > b and 0 if a < b.
   * (if a = b, it doesn't matter what value k is) */

  // If a and b have different signs, then k = sign(a)
  int use_sign_of_a = sa ^ sb;

  // If a and b have the same sign, then k = sign(a - b)
  int use_sign_of_c = flip(sa ^ sb);

  int k = use_sign_of_a * sa + use_sign_of_c * sc;
  int q = flip(k); // opposite of k

  return a * k + b * q;
}
```

Note that for clarity, we split up the code into many different methods and variables. This is certainly not the most compact or efficient way to write it, but it does make what we're doing much cleaner.

16.8 **English Int:** Given any integer, print an English phrase that describes the integer (e.g., "One Thousand, Two Hundred Thirty Four").

pg 182

SOLUTION

This is not an especially challenging problem, but it is a somewhat tedious one. The key is to be organized in how you approach the problem—and to make sure you have good test cases.

We can think about converting a number like 19,323,984 as converting each of three 3-digit segments of the number, and inserting "thousands" and "millions" in between as appropriate. That is,

```
convert(19,323,984) = convert(19) + " million " + convert(323) + " thousand " +
                      convert(984)
```

The code below implements this algorithm.

```
String[] smalls = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
  "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
  "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
String[] tens = {"", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy",
  "Eighty", "Ninety"};
String[] bigs = {"", "Thousand", "Million", "Billion"};
String hundred = "Hundred";
String negative = "Negative";

String convert(int num) {
  if (num == 0) {
    return smalls[0];
  } else if (num < 0) {
    return negative + " " + convert(-1 * num);
  }

  LinkedList<String> parts = new LinkedList<String>();
  int chunkCount = 0;

  while (num > 0) {
    if (num % 1000 != 0) {
      String chunk = convertChunk(num % 1000) + " " + bigs[chunkCount];
      parts.addFirst(chunk);
    }
    num /= 1000; // shift chunk
    chunkCount++;
  }

  return listToString(parts);
}

String convertChunk(int number) {
  LinkedList<String> parts = new LinkedList<String>();

  /* Convert hundreds place */
  if (number >= 100) {
    parts.addLast(smalls[number / 100]);
```

```
      parts.addLast(hundred);
      number %= 100;
   }

   /* Convert tens place */
   if (number >= 10 && number <= 19) {
      parts.addLast(smalls[number]);
   } else if (number >= 20) {
      parts.addLast(tens[number / 10]);
      number %= 10;
   }

   /* Convert ones place */
   if (number >= 1 && number <= 9) {
      parts.addLast(smalls[number]);
   }

   return listToString(parts);
}
/* Convert a linked list of strings to a string, dividing it up with spaces. */
String listToString(LinkedList<String> parts) {
   StringBuilder sb = new StringBuilder();
   while (parts.size() > 1) {
      sb.append(parts.pop());
      sb.append(" ");
   }
   sb.append(parts.pop());
   return sb.toString();
}
```

The key in a problem like this is to make sure you consider all the special cases. There are a lot of them.

16.9 Operations: Write methods to implement the multiply, subtract, and divide operations for integers. The results of all of these are integers. Use only the add operator.

pg 182

SOLUTION

The only operation we have to work with is the add operator. In each of these problems, it's useful to think in depth about what these operations really do or how to phrase them in terms of other operations (either add or operations we've already completed).

Subtraction

How can we phrase subtraction in terms of addition? This one is pretty straightforward. The operation `a - b` is the same thing as `a + (-1) * b`. However, because we are not allowed to use the * (multiply) operator, we must implement a `negate` function.

```
/* Flip a positive sign to negative or negative sign to pos. */
int negate(int a) {
   int neg = 0;
   int newSign = a < 0 ? 1 : -1;
   while (a != 0) {
      neg += newSign;
      a += newSign;
   }
```

```
    return neg;
}

/* Subtract two numbers by negating b and adding them */
int minus(int a, int b) {
    return a + negate(b);
}
```

The negation of the value k is implemented by adding -1 k times. Observe that this will take `O(k)` time.

If optimizing is something we value here, we can try to get a to zero faster. (For this explanation, we'll assume that a is positive.) To do this, we can first reduce a by 1, then 2, then 4, then 8, and so on. We'll call this value `delta`. We want a to reach exactly zero. When reducing a by the next `delta` would change the sign of a, we reset `delta` back to 1 and repeat the process.

For example:

```
a:      29   28   26   22   14   13   11    7    6    4    0
delta:     -1   -2   -4   -8   -1   -2   -4   -1   -2   -4
```

The code below implements this algorithm.

```
int negate(int a) {
    int neg = 0;
    int newSign = a < 0 ? 1 : -1;
    int delta = newSign;
    while (a != 0) {
        boolean differentSigns = (a + delta > 0) != (a > 0);
        if (a + delta != 0 && differentSigns) { // If delta is too big, reset it.
            delta = newSign;
        }
        neg += delta;
        a += delta;
        delta += delta; // Double the delta
    }
    return neg;
}
```

Figuring out the runtime here takes a bit of calculation.

Observe that reducing a by half takes `O(log a)` work. Why? For each round of "reduce a by half", the absolute values of a and `delta` always add up to the same number. The values of `delta` and a will converge at $a/2$. Since `delta` is being doubled each time, it will take `O(log a)` steps to reach half of a.

We do O(log a) rounds.

1. Reducing a to $a/2$ takes `O(log a)` time.
2. Reducing $a/2$ to $a/4$ takes `O(log` $a/2$`)` time.
3. Reducing $a/4$ to $a/8$ takes `O(log` $a/4$`)` time.

... As so on, for `O(log a)` rounds.

The runtime therefore is $O(\log a + \log(a/2) + \log(a/4) + \ldots)$, with `O(log a)` terms in the expression.

Recall two rules of logs:

- $\log(xy) = \log x + \log y$
- $\log(x/y) = \log x - \log y$.