```
            if (str.charAt(i) == '(') {
               String s = insertInside(str, i);
               /* Add s to set if it's not already in there. Note:  HashSet
                * automatically checks for duplicates before adding, so an explicit
                * check is not necessary. */
               set.add(s);
            }
         }
         set.add("()" + str);
      }
   }
   return set;
}

String insertInside(String str, int leftIndex) {
   String left = str.substring(0, leftIndex + 1);
   String right = str.substring(leftIndex + 1, str.length());
   return left + "()" + right;
}
```

This works, but it's not very efficient. We waste a lot of time coming up with the duplicate strings.

We can avoid this duplicate string issue by building the string from scratch. Under this approach, we add left and right parens, as long as our expression stays valid.

On each recursive call, we have the index for a particular character in the string. We need to select either a left or a right paren. When can we use a left paren, and when can we use a right paren?

1. *Left Paren:* As long as we haven't used up all the left parentheses, we can always insert a left paren.
2. *Right Paren:* We can insert a right paren as long as it won't lead to a syntax error. When will we get a syntax error? We will get a syntax error if there are more right parentheses than left.

So, we simply keep track of the number of left and right parentheses allowed. If there are left parens remaining, we'll insert a left paren and recurse. If there are more right parens remaining than left (i.e., if there are more left parens in use than right parens), then we'll insert a right paren and recurse.

```
void addParen(ArrayList<String> list, int leftRem, int rightRem, char[] str,
              int index) {
   if (leftRem < 0 || rightRem < leftRem) return; // invalid state

   if (leftRem == 0 && rightRem == 0) { /* Out of left and right parentheses */
      list.add(String.copyValueOf(str));
   } else {
      str[index] = '('; // Add left and recurse
      addParen(list, leftRem - 1, rightRem, str, index + 1);

      str[index] = ')'; // Add right and recurse
      addParen(list, leftRem, rightRem - 1, str, index + 1);
   }
}

ArrayList<String> generateParens(int count) {
   char[] str = new char[count*2];
   ArrayList<String> list = new ArrayList<String>();
   addParen(list, count, count, str, 0);
   return list;
}
```

Because we insert left and right parentheses at each index in the string, and we never repeat an index, each string is guaranteed to be unique.

8.10 Paint Fill: Implement the "paint fill" function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

pg 136

SOLUTION

First, let's visualize how this method works. When we call `paintFill` (i.e., "click" paint fill in the image editing application) on, say, a green pixel, we want to "bleed" outwards. Pixel by pixel, we expand outwards by calling `paintFill` on the surrounding pixel. When we hit a pixel that is not green, we stop.

We can implement this algorithm recursively:

```
1  enum Color { Black, White, Red, Yellow, Green }
2
3  boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
4     if (screen[r][c] == ncolor) return false;
5     return PaintFill(screen, r, c, screen[r][c], ncolor);
6  }
7
8  boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
9     if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
10       return false;
11    }
12
13    if (screen[r][c] == ocolor) {
14       screen[r][c] = ncolor;
15       PaintFill(screen, r - 1, c, ocolor, ncolor); // up
16       PaintFill(screen, r + 1, c, ocolor, ncolor); // down
17       PaintFill(screen, r, c - 1, ocolor, ncolor); // left
18       PaintFill(screen, r, c + 1, ocolor, ncolor); // right
19    }
20    return true;
21 }
```

If you used the variable names `x` and `y` to implement this, be careful about the ordering of the variables in `screen[y][x]`. Because `x` represents the *horizontal* axis (that is, it's left to right), it actually corresponds to the column number, not the row number. The value of `y` equals the number of rows. This is a very easy place to make a mistake in an interview, as well as in your daily coding. It's typically clearer to use `row` and `column` instead, as we've done here.

Does this algorithm seem familiar? It should! This is essentially depth-first search on a graph. At each pixel, we are searching outwards to each surrounding pixel. We stop once we've fully traversed all the surrounding pixels of this color.

We could alternatively implement this using breadth-first search.

8.11 Coins: Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

pg 136

SOLUTION

This is a recursive problem, so let's figure out how to compute `makeChange(n)` using prior solutions (i.e., subproblems).

Let's say `n = 100`. We want to compute the number of ways of making change for 100 cents. What is the relationship between this problem and its subproblems?

We know that making change for 100 cents will involve either 0, 1, 2, 3, or 4 quarters. So:

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(100 using 1 quarter)  +
                  makeChange(100 using 2 quarters) +
                  makeChange(100 using 3 quarters) +
                  makeChange(100 using 4 quarters)
```

Inspecting this further, we can see that some of these problems reduce. For example, `makeChange(100 using 1 quarter)` will equal `makeChange(75 using 0 quarters)`. This is because, if we must use exactly one quarter to make change for 100 cents, then our only remaining choices involve making change for the remaining 75 cents.

We can apply the same logic to `makeChange(100 using 2 quarters)`, `makeChange(100 using 3 quarters)` and `makeChange(100 using 4 quarters)`. We have thus reduced the above statement to the following.

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(75 using 0 quarters) +
                  makeChange(50 using 0 quarters) +
                  makeChange(25 using 0 quarters) +
                  1
```

Note that the final statement from above, `makeChange(100 using 4 quarters)`, equals 1. We call this "fully reduced."

Now what? We've used up all our quarters, so now we can start applying our next biggest denomination: dimes.

Our approach for quarters applies to dimes as well, but we apply this for *each* of the four of five parts of the above statement. So, for the first part, we get the following statements:

```
makeChange(100 using 0 quarters) = makeChange(100 using 0 quarters, 0 dimes) +
                                   makeChange(100 using 0 quarters, 1 dime)  +
                                   makeChange(100 using 0 quarters, 2 dimes) +
                                   ...
                                   makeChange(100 using 0 quarters, 10 dimes)

makeChange(75 using 0 quarters) = makeChange(75 using 0 quarters, 0 dimes) +
                                  makeChange(75 using 0 quarters, 1 dime)  +
                                  makeChange(75 using 0 quarters, 2 dimes) +
                                  ...
                                  makeChange(75 using 0 quarters, 7 dimes)

makeChange(50 using 0 quarters) = makeChange(50 using 0 quarters, 0 dimes) +
                                  makeChange(50 using 0 quarters, 1 dime)  +
                                  makeChange(50 using 0 quarters, 2 dimes) +
```

```
                                    ...
                                    makeChange(50 using 0 quarters, 5 dimes)

makeChange(25 using 0 quarters) = makeChange(25 using 0 quarters, 0 dimes) +
                                  makeChange(25 using 0 quarters, 1 dime)  +
                                  makeChange(25 using 0 quarters, 2 dimes)
```

Each one of these, in turn, expands out once we start applying nickels. We end up with a tree-like recursive structure where each call expands out to four or more calls.

The base case of our recursion is the fully reduced statement. For example, `makeChange(50 using 0 quarters, 5 dimes)` is fully reduced to 1, since 5 dimes equals 50 cents.

This leads to a recursive algorithm that looks like this:

```
1  int makeChange(int amount, int[] denoms, int index) {
2    if (index >= denoms.length - 1) return 1; // last denom
3    int denomAmount = denoms[index];
4    int ways = 0;
5    for (int i = 0; i * denomAmount <= amount; i++) {
6      int amountRemaining = amount - i * denomAmount;
7      ways += makeChange(amountRemaining, denoms, index + 1);
8    }
9    return ways;
10 }
11
12 int makeChange(int n) {
13   int[] denoms = {25, 10, 5, 1};
14   return makeChange(n, denoms, 0);
15 }
```

This works, but it's not as optimal as it could be. The issue is that we will be recursively calling `makeChange` several times for the same values of `amount` and `index`.

We can resolve this issue by storing the previously computed values. We'll need to store a mapping from each pair `(amount, index)` to the precomputed result.

```
1  int makeChange(int n) {
2    int[] denoms = {2  510, 5, 1};
3    int[][] map = new int[n + 1][denoms.length]; // precomputed vals
4    return makeChange(n, denoms, 0, map);
5  }
6
7  int makeChange(int amount, int[] denoms, int index, int[][] map) {
8    if (map[amount][index] > 0) { // retrieve value
9      return map[amount][index];
10   }
11   if (index >= denoms.length - 1) return 1; // one denom remaining
12   int denomAmount = denoms[index];
13   int ways = 0;
14   for (int i = 0; i * denomAmount <= amount; i++) {
15     // go to next denom, assuming i coins of denomAmount
16     int amountRemaining = amount - i * denomAmount;
17     ways += makeChange(amountRemaining, denoms, index + 1, map);
18   }
19   map[amount][index] = ways;
20   return ways;
21 }
```

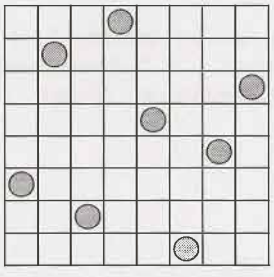
Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from `amount` *to a new hash table, which then maps from* `denom` *to the precomputed value. There are other alternative data structures as well.*

8.12 **Eight Queens:** Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

pg 136

SOLUTION

We have eight queens which must be lined up on an 8x8 chess board such that none share the same row, column or diagonal. So, we know that each row and column (and diagonal) must be used exactly once.



A "Solved" Board with 8 Queens

Picture the queen that is placed last, which we'll assume is on row 8. (This is an okay assumption to make since the ordering of placing the queens is irrelevant.) On which cell in row 8 is this queen? There are eight possibilities, one for each column.

So if we want to know all the valid ways of arranging 8 queens on an 8x8 chess board, it would be:

```
ways to arrange 8 queens on an 8x8 board =
   ways to arrange 8 queens on an 8x8 board with queen at (7, 0) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 1) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 2) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 3) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 4) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 5) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 6) +
   ways to arrange 8 queens on an 8x8 board with queen at (7, 7)
```

We can compute each one of these using a very similar approach:

```
ways to arrange 8 queens on an 8x8 board with queen at (7, 3) =
   ways to ... with queens at (7, 3) and (6, 0) +
   ways to ... with queens at (7, 3) and (6, 1) +
   ways to ... with queens at (7, 3) and (6, 2) +
   ways to ... with queens at (7, 3) and (6, 4) +
   ways to ... with queens at (7, 3) and (6, 5) +
   ways to ... with queens at (7, 3) and (6, 6) +
   ways to ... with queens at (7, 3) and (6, 7)
```

Note that we don't need to consider combinations with queens at `(7, 3)` and `(6, 3)`, since this is a violation of the requirement that every queen is in its own row, column and diagonal.

Implementing this is now reasonably straightforward.

```
int GRID_SIZE = 8;

void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
   if (row == GRID_SIZE) { // Found valid placement
      results.add(columns.clone());
   } else {
      for (int col = 0; col < GRID_SIZE; col++) {
         if (checkValid(columns, row, col)) {
            columns[row] = col;  // Place queen
            placeQueens(row + 1, columns, results);
         }
      }
   }
}

/* Check if (row1, column1) is a valid spot for a queen by checking if there is a
 * queen in the same column or diagonal. We don't need to check it for queens in
 * the same row because the calling placeQueen only attempts to place one queen at
 * a time. We know this row is empty. */
boolean checkValid(Integer[] columns, int row1, int column1) {
   for (int row2 = 0; row2 < row1; row2++) {
      int column2 = columns[row2];
      /* Check if (row2, column2) invalidates (row1, column1) as a
       * queen spot. */

      /* Check if rows have a queen in the same column */
      if (column1 == column2) {
         return false;
      }

      /* Check diagonals: if the distance between the columns equals the distance
       * between the rows, then they're in the same diagonal. */
      int columnDistance = Math.abs(column2 - column1);

      /* row1 > row2, so no need for abs */
      int rowDistance = row1 - row2;
      if (columnDistance == rowDistance) {
         return false;
      }
   }
   return true;
}
```

Observe that since each row can only have one queen, we don't need to store our board as a full 8x8 matrix. We only need a single array where `column[r] = c` indicates that row `r` has a queen at column `c`.

8.13 Stack of Boxes: You have a stack of n boxes, with widths w_i, heights h_i, and depths d_i. The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

pg 136

SOLUTION

To tackle this problem, we need to recognize the relationship between the different subproblems.

Solution #1

Imagine we had the following boxes: $b_1, b_2, \ldots, b_n$. The biggest stack that we can build with all the boxes equals the max of (biggest stack with bottom b_1, biggest stack with bottom b_2, ..., biggest stack with bottom b_n). That is, if we experimented with each box as a bottom and built the biggest stack possible with each, we would find the biggest stack possible.

But, how would we find the biggest stack with a particular bottom? Essentially the same way. We experiment with different boxes for the second level, and so on for each level.

Of course, we only experiment with valid boxes. If b_5 is bigger than b_1, then there's no point in trying to build a stack that looks like $\{b_1, b_5, \ldots\}$. We already know b_1 can't be below b_5.

We can perform a small optimization here. The requirements of this problem stipulate that the lower boxes must be strictly greater than the higher boxes in all dimensions. Therefore, if we sort (descending order) the boxes on a dimension—any dimension—then we know we don't have to look backwards in the list. The box b_1 cannot be on top of box b_5, since its height (or whatever dimension we sorted on) is greater than b_5's height.

The code below implements this algorithm recursively.

```
1  int createStack(ArrayList<Box> boxes) {
2    /* Sort in decending order by height. */
3    Collections.sort(boxes, new BoxComparator());
4    int maxHeight = 0;
5    for (int i = 0; i < boxes.size(); i++) {
6      int height = createStack(boxes, i);
7      maxHeight = Math.max(maxHeight, height);
8    }
9    return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex) {
13   Box bottom = boxes.get(bottomIndex);
14   int maxHeight = 0;
15   for (int i = bottomIndex + 1; i < boxes.size(); i++) {
16     if (boxes.get(i).canBeAbove(bottom)) {
17       int height = createStack(boxes, i);
18       maxHeight = Math.max(height, maxHeight);
19     }
20   }
21   maxHeight += bottom.height;
22   return maxHeight;
23 }
24
25 class BoxComparator implements Comparator<Box> {
```

```
26    @Override
27    public int compare(Box x, Box y){
28       return y.height - x.height;
29    }
30 }
```

The problem in this code is that it gets very inefficient. We try to find the best solution that looks like $\{b_3, b_4, \ldots\}$ even though we may have already found the best solution with b_4 at the bottom. Instead of generating these solutions from scratch, we can cache these results using memoization.

```
1  int createStack(ArrayList<Box> boxes) {
2     Collections.sort(boxes, new BoxComparator());
3     int maxHeight = 0;
4     int[] stackMap = new int[boxes.size()];
5     for (int i = 0; i < boxes.size(); i++) {
6        int height = createStack(boxes, i, stackMap);
7        maxHeight = Math.max(maxHeight, height);
8     }
9     return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
13    if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
14       return stackMap[bottomIndex];
15    }
16
17    Box bottom = boxes.get(bottomIndex);
18    int maxHeight = 0;
19    for (int i = bottomIndex + 1; i < boxes.size(); i++) {
20       if (boxes.get(i).canBeAbove(bottom)) {
21          int height = createStack(boxes, i, stackMap);
22          maxHeight = Math.max(height, maxHeight);
23       }
24    }
25    maxHeight += bottom.height;
26    stackMap[bottomIndex] = maxHeight;
27    return maxHeight;
28 }
```

Because we're only mapping from an index to a height, we can just use an integer array for our "hash table."

Be very careful here with what each spot in the hash table represents. In this code, `stackMap[i]` represents the tallest stack with box `i` at the bottom. Before pulling the value from the hash table, you have to ensure that box `i` can be placed on top of the current bottom.

It helps to keep the line that recalls from the hash table symmetric with the one that inserts. For example, in this code, we recall from the hash table with `bottomIndex` at the start of the method. We insert into the hash table with `bottomIndex` at the end.

Solution #2

Alternatively, we can think about the recursive algorithm as making a choice, at each step, whether to put a particular box in the stack. (We will again sort our boxes in descending order by a dimension, such as height.)

First, we choose whether or not to put box 0 in the stack. Take one recursive path with box 0 at the bottom and one recursive path without box 0. Return the better of the two options.

Then, we choose whether or not to put box 1 in the stack. Take one recursive path with box 1 at the bottom and one path without box 1. Return the better of the two options.

We will again use memoization to cache the height of the tallest stack with a particular bottom.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int[] stackMap = new int[boxes.size()];
4    return createStack(boxes, null, 0, stackMap);
5  }
6
7  int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
8    if (offset >= boxes.size()) return 0; // Base case
9
10   / *height with this bottom */
11   Box newBottom = boxes.get(offset);
12   int heightWithBottom = 0;
13   if (bottom == null || newBottom.canBeAbove(bottom)) {
14     if (stackMap[offset] == 0) {
15       stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16       stackMap[offset] += newBottom.height;
17     }
18     heightWithBottom = stackMap[offset];
19   }
20
21   / *without this bottom */
22   int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24   /* Return better of two options. */
25   return Math.max(heightWithBottom, heightWithoutBottom);
26 }
```

Again, pay close attention to when you recall and insert values into the hash table. It's typically best if these are symmetric, as they are in lines 15 and 16-18.

8.14 Boolean Evaluation: Given a boolean expression consisting of the symbols `0` (false), `1` (true), `&` (AND), `|` (OR), and `^` (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`. The expression should be fully parenthesized (e.g., `(0)^(1)`) but not extraneously (e.g., `(((0))^(1))`).

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

pg 136

SOLUTION

As in other recursive problems, the key to this problem is to figure out the relationship between a problem and its subproblems.

Brute Force

Consider an expression like `0^0&0^1|1` and the target result `true`. How can we break down `countEval(0^0&0^1|1, true)` into smaller problems?

We could just essentially iterate through each possible place to put a parenthesis.

```
countEval(0^0&0^1|1, true) =
   countEval(0^0&0^1|1 where paren around char 1, true)
 + countEval(0^0&0^1|1 where paren around char 3, true)
 + countEval(0^0&0^1|1 where paren around char 5, true)
 + countEval(0^0&0^1|1 where paren around char 7, true)
```

Now what? Let's look at just one of those expressions—the paren around char 3. This gives us `(0^0)&(0^1)`.

In order to make that expression true, both the left and right sides must be true. So:

```
left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)
```

The reason we multiply the results of the left and right sides is that each result from the two sides can be paired up with each other to form a unique combination.

Each of those terms can now be decomposed into smaller problems in a similar process.

What happens when we have an "|" (OR)? Or an "^" (XOR)?

If it's an OR, then either the left or the right side must be true—or both.

```
countEval(left | right, true) = countEval(left, true)  * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
                              + countEval(left, true)  * countEval(right, true)
```

If it's an XOR, then the left or the right side can be true, but not both.

```
countEval(left ^ right, true) = countEval(left, true) * countEval(right, false)
                              + countEval(left, false) * countEval(right, true)
```

What if we were trying to make the result `false` instead? We can switch up the logic from above:

```
countEval(left & right, false) = countEval(left, true)  * countEval(right, false)
                               + countEval(left, false) * countEval(right, true)
                               + countEval(left, false) * countEval(right, false)
countEval(left | right, false) = countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) = countEval(left, false) * countEval(right, false)
                               + countEval(left, true)  * countEval(right, true)
```

Alternatively, we can just use the same logic from above and subtract it out from the total number of ways of evaluating the expression.

```
totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)
```

This makes the code a bit more concise.

```
1  int countEval(String s, boolean result) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
4
5    int ways = 0;
6    for (int i = 1; i < s.length(); i += 2) {
7      char c = s.charAt(i);
8      String left = s.substring(0, i);
9      String right = s.substring(i + 1, s.length());
10
11     /* Evaluate each side for each result. */
12     int leftTrue = countEval(left, true);
13     int leftFalse = countEval(left, false);
14     int rightTrue = countEval(right, true);
```