# Recursion and Dynamic Programming

*Concepts and Algorithms: Solutions*

**Chapter 9**

**9.1** *A child is running up a staircase with n steps, and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.*

*pg 109*

## SOLUTION

We can approach this problem from the top down. On the very last hop, up to the nth step, the child could have done either a single, double, or triple step hop. That is, the last move might have been a single step hop from step `n-1`, a double step hop from step `n-2`, or a triple step hop from `n-3`. The total number of ways of reaching the last step is therefore the sum of the number of ways of reaching each of the last three steps.

A simple implementation of this code is below.

```
1  public int countWays(int n) {
2     if (n < 0) {
3        return 0;
4     } else if (n == 0) {
5        return 1;
6     } else {
7        return countWays(n - 1) + countWays(n - 2) +
8               countWays(n - 3);
9     }
10 }
```

Like the Fibonacci problem, the runtime of this algorithm is exponential (specifically, $O(3^N)$), since each call branches out to three more calls. This means that `countWays` is called many times for the same values, which is unnecessary. We can fix this through dynamic programming.

```
11 public static int countWaysDP(int n, int[] map) {
12    if (n < 0) {
13       return 0;
14    } else if (n == 0) {
15       return 1;
16    } else if (map[n] > -1) {
17       return map[n];
18    } else {
19       map[n] = countWaysDP(n - 1, map) +
20                countWaysDP(n - 2, map) +
21                countWaysDP(n - 3, map);
22       return map[n];
23    }
24 }
```

Regardless of whether or not you use dynamic programming, note that the number of ways will quickly overflow the bounds of an integer. By the time you get to just `n = 37`, the result has already overflowed. Using a `long` will delay, but not completely solve, this issue.

**9.2** *Imagine a robot sitting on the upper left corner of an X by Y grid. The robot can only move in two directions: right and down. How many possible paths are there for the robot to go from (0,0) to (X,Y)?*

*FOLLOW UP*

*Imagine certain spots are "off limits," such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.*

*pg 109*

**SOLUTION**

We need to count the number of ways of making a path with X right steps and Y down steps. This path will have X+Y steps total.

To build a path, we are essentially selecting X times to move right out of a total of X+Y moves. Thus, the number of total paths must be the number of ways of selecting X items out of X+Y items. This is given by the binomial expression (a.k.a., "n choose r"):

$$\binom{n}{r} = \frac{n!}{r!\,(n-r)!}$$

In terms of this problem, the expression is:

$$\binom{X+Y}{X} = \frac{(X+Y)!}{X!\,Y!}$$

If you didn't know the binomial expression, you could still deduce how to solve this problem.

Think about each path as a string of length X+Y consisting X `'R'`s and Y `'D'`s. We know that the number of strings we can make with X+Y *unique* characters is `(X+Y)!`. However, in this case, X of the characters are `'R'`s and Y are `'D'`s. Since the `'R'`s can be rearranged in `X!` ways, each of which is identical, and we can do an equivalent thing with the `'D'`s, we need to divide out by `X!` and `Y!`. We then get the same expression as we had before:

$$\frac{(X+Y)!}{X!\,Y!}$$

**Follow Up: Find a path (with off limit spots)**

If we picture our grid, the only way to move to spot `(X,Y)` is by moving to one of the adjacent spots: `(X-1,Y)` or `(X,Y-1)`. So, we need to find a path to either `(X-1,Y)` or `(X,Y-1)`.

How do we find a path to those spots? To find a path to `(X-1,Y)` or `(X,Y-1)`, we need to move to one of its adjacent cells. So, we need to find a path to a spot adjacent to `(X-1,Y)`, which are coordinates `(X-2,Y)` and `(X-1,Y-1)`, or a spot adjacent to

(X,Y-1), which are spots (X-1,Y-1) and (X,Y-2). Observe that we list the point (X-1,Y-1) twice; we'll discuss that issue later.

So then, to find a path from the origin, we just work backwards like this. Starting from the last cell, we try to find a path to each of its adjacent cells. The recursive code below implements this algorithm.

```
1  public boolean getPath(int x, int y, ArrayList<Point> path) {
2    Point p = new Point(x, y);
3    path.add(p);
4    if (x == 0 && y == 0) {
5      return true; // found a path
6    }
7    boolean success = false;
8    if (x >= 1 && isFree(x - 1, y)) { // Try left
9      success = getPath(x - 1, y, path); // Free! Go left
10   }
11   if (!success && y >= 1 && isFree(x, y - 1)) { // Try up
12     success = getPath(x, y - 1, path); // Free! Go up
13   }
14   if (success) {
15     path.add(p); // Right way! Add to path.
16   }
17   return success;
18 }
```

Earlier, we'd mentioned an issue with duplicate paths. To find a path to (X,Y), we look for a path to an adjacent coordinate: (X-1,Y) or (X,Y-1). Of course, if one of those squares is off limits, we ignore it. Then, we look at their adjacent coordinates: (X-2,Y), (X-1,Y-1), (X-1,Y-1), and (X,Y-2). The spot (X-1,Y-1) appears twice, which means that we're duplicating effort. Ideally, we should remember that we already visited (X-1,Y-1) so that we don't waste our time.

This is what the dynamic programming algorithm below does.

```
1  public boolean getPath(int x, int y, ArrayList<Point> path,
2                         Hashtable<Point, Boolean> cache) {
3    Point p = new Point(x, y);
4    if (cache.containsKey(p)) { // Already visited this cell
5      return cache.get(p);
6    }
7    if (x == 0 && y == 0) {
8      return true; // found a path
9    }
10   boolean success = false;
11   if (x >= 1 && isFree(x - 1, y)) { // Try right
12     success = getPath(x - 1, y, path, cache); // Free!  Go right
13   }
14   if (!success && y >= 1 && isFree(x, y - 1)) { // Try down
15     success = getPath(x, y - 1, path, cache); // Free!  Go down
16   }
17   if (success) {
```

```
18        path.add(p); // Right way! Add to path
19     }
20     cache.put(p, success); // Cache result
21     return success;
22  }
```

This simple change will make our code run substantially faster.

**9.3** *A magic index in an array A[1...n-1] is defined to be an index such that A[i] = i. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.*

*FOLLOW UP*

*What if the values are not distinct?*

*pg 109*

**SOLUTION**

Immediately, the brute force solution should jump to mind—and there's no shame in mentioning it. We simply iterate through the array, looking for an element which matches this condition.

```
1  public static int magicSlow(int[] array) {
2     for (int i = 0; i < array.length; i++) {
3        if (array[i] == i) {
4           return i;
5        }
6     }
7     return -1;
8  }
```

Given that the array is sorted though, it's very likely that we're supposed to use this condition.

We may recognize that this problem sounds a lot like the classic binary search problem. Leveraging the Pattern Matching approach for generating algorithms, how might we apply binary search here?

In binary search, we find an element k by comparing it to the middle element, x, and determining if k would land on the left or the right side of x.

Building off this approach, is there a way that we can look at the middle element to determine where a magic index might be? Let's look at a sample array:

| -40 | -20 | -1 | 1 | 2 | 3 | 5 | 7 | 9 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

When we look at the middle element A[5] = 3, we know that the magic index must be on the right side, since A[mid] < mid.

Why couldn't the magic index be on the left side? Observe that when we move from `i` to `i-1`, the value at this index must decrease by at least 1, if not more (since the array is sorted and all the elements are distinct). So, if the middle element is already too small to be a magic index, then when we move to the left, subtracting k indexes and (at least) k values, all subsequent elements will also be too small.

We continue to apply this recursive algorithm, developing code that looks very much like binary search.

```
1  public static int magicFast(int[] array, int start, int end) {
2     if (end < start || start < 0 || end >= array.length) {
3        return -1;
4     }
5     int mid = (start + end) / 2;
6     if (array[mid] == mid) {
7        return mid;
8     } else if (array[mid] > mid){
9        return magicFast(array, start, mid - 1);
10    } else {
11       return magicFast(array, mid + 1, end);
12    }
13 }
14
15 public static int magicFast(int[] array) {
16    return magicFast(array, 0, array.length - 1);
17 }
```

**Follow Up: What if the elements are not distinct?**

If the elements are not distinct, then this algorithm fails. Consider the following array:

| -10 | -5 | 2 | 2 | 2 | 3 | 4 | 7 | 9 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

When we see that `A[mid] < mid`, we cannot conclude which side the magic index is on. It could be on the right side, as before. Or, it could be on the left side (as it, in fact, is).

Could it be *anywhere* on the left side? Not exactly. Since `A[5] = 3`, we know that `A[4]` couldn't be a magic index. `A[4]` would need to be 4 to be the magic index, but `A[4]` must be less than `A[5]`.

In fact, when we see that `A[5] = 3`, we'll need to recursively search the right side as before. But, to search the left side, we can skip a bunch of elements and only recursively search elements `A[0]` through `A[3]`. `A[3]` is the first element that could be a magic index.

The general pattern is that we compare `midIndex` and `midValue` for equality first. Then, if they are not equal, we recursively search the left and right sides as follows:

- Left side: search indices `start` through `Math.min(midIndex - 1, midValue)`.

- Right side: search indices `Math.max(midIndex + 1, midValue)` through end.

The code below implements this algorithm.

```
1  public static int magicFast(int[] array, int start, int end) {
2     if (end < start || start < 0 || end >= array.length) {
3        return -1;
4     }
5     int midIndex = (start + end) / 2;
6     int midValue = array[midIndex];
7     if (midValue == midIndex) {
8        return midIndex;
9     }
10
11    /* Search left */
12    int leftIndex = Math.min(midIndex - 1, midValue);
13    int left = magicFast(array, start, leftIndex);
14    if (left >= 0) {
15       return left;
16    }
17
18    /* Search right */
19    int rightIndex = Math.max(midIndex + 1, midValue);
20    int right = magicFast(array, rightIndex, end);
21
22    return right;
23 }
24
25 public static int magicFast(int[] array) {
26    return magicFast(array, 0, array.length - 1);
27 }
```

Note that in the above code, if the elements are all distinct, the method operates almost identically to the first solution.

**9.4** *Write a method to return all subsets of a set.*

*pg 109*

## SOLUTION

We should first have some reasonable expectations of our time and space complexity. How many subsets of a set are there? We can compute this by realizing that when we generate a subset, each element has the "choice" of either being in there or not. That is, for the first element, there are two choices: it is either in the set, or it is not. For the second, there are two, etc. So, doing `{2 * 2 * ...}` n times gives us $2^n$ subsets. We will therefore not be able to do better than $O(2^n)$ in time or space complexity.

The subsets of $\{a_1, a_2, \ldots, a_n\}$ are also called the powerset, $P(\{a_1, a_2, \ldots, a_n\})$, or just $P(n)$.

**Solution #1: Recursion**

This problem is a good candidate for the Base Case and Build approach. Imagine that we are trying to find all subsets of a set like S = $\{a_1, a_2, \ldots, a_n\}$. We can start with the Base Case.

*Base Case:* n = 0.

There is just one subset of the empty set: {}.

*Case:* n = 1.

There are two subsets of the set $\{a_1\}$: {}, $\{a_1\}$.

*Case:* n = 2.

There are four subsets of the set $\{a_1, a_2\}$: {}, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$.

*Case:* n = 3.

Now here's where things get interesting. We want to find a way of generating the solution for n = 3 based on the prior solutions.

What is the difference between the solution for n = 3 and the solution for n = 2? Let's look at this more deeply:

P(2) = {}, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$
P(3) = {}, $\{a_1\}$, $\{a_2\}$, $\{a_3\}$, $\{a_1, a_2\}$, $\{a_1, a_3\}$, $\{a_2, a_3\}$, $\{a_1, a_2, a_3\}$

The difference between these solutions is that P(2) is missing all the subsets containing $a_3$.

P(3) - P(2) = $\{a_3\}$, $\{a_1, a_3\}$, $\{a_2, a_3\}$, $\{a_1, a_2, a_3\}$

How can we use P(2) to create P(3)? We can simply clone the subsets in P(2) and add $a_3$ to them:

P(2) = {}, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$
P(2) + $a_3$ = $\{a_3\}$, $\{a_1, a_3\}$, $\{a_2, a_3\}$, $\{a_1, a_2, a_3\}$

When merged together, the lines above make P(3).

*Case:* n > 0

Generating P(n) for the general case is just a simple generalization of the above steps. We compute P(n-1), clone the results, and then add $a_n$ to each of these cloned sets.

The following code implements this algorithm:

```
1  ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set,
2                                           int index) {
3     ArrayList<ArrayList<Integer>> allsubsets;
4     if (set.size() == index) { // Base case - add empty set
5        allsubsets = new ArrayList<ArrayList<Integer>>();
6        allsubsets.add(new ArrayList<Integer>()); // Empty set
```

```
7     } else {
8        allsubsets = getSubsets(set, index + 1);
9        int item = set.get(index);
10       ArrayList<ArrayList<Integer>> moresubsets =
11          new ArrayList<ArrayList<Integer>>();
12       for (ArrayList<Integer> subset : allsubsets) {
13          ArrayList<Integer> newsubset = new ArrayList<Integer>();
14          newsubset.addAll(subset); //
15          newsubset.add(item);
16          moresubsets.add(newsubset);
17       }
18       allsubsets.addAll(moreśubsets);
19    }
20    return allsubsets;
21 }
```

This solution will be $O(2^n)$ in time and space, which is the best we can do. For a slight optimization, we could also implement this algorithm iteratively.

### Solution #2: Combinatorics

While there's nothing wrong with the above solution, there's another way to approach it.

Recall that when we're generating a set, we have two choices for each element: (1) the element is in the set (the "yes" state) or (2) the element is not in the set (the "no" state). This means that each subset is a secuence of yeses / nos—e.g., "yes, yes, no, no, yes, no"

This gives us $2^n$ possible subsets. How can we iterate through all possible sequences of "yes" / "no" states for all elements? If each "yes" can be treated as a 1 and each "no" can be treated as a 0, then each subset can be represented as a binary string.

Generating all subsets, then, really just comes down to generating all binary numbers (that is, all integers). We iterate through all numbers from 1 to $2^n$ and translate the binary representation of the numbers into a set. Easy!

```
1  ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
2     ArrayList<ArrayList<Integer>> allsubsets =
3        new ArrayList<ArrayList<Integer>>();
4     int max = 1 << set.size(); /* Compute 2^n */
5     for (int k = 0; k < max; k++) {
6        ArrayList<Integer> subset = convertIntToSet(k, set);
7        allsubsets.add(subset);
8     }
9     return allsubsets;
10 }
11
12 ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
13    ArrayList<Integer> subset = new ArrayList<Integer>();
14    int index = 0;
```

```
15    for (int k = x; k > 0; k >>= 1) {
16       if ((k & 1) == 1) {
17          subset.add(set.get(index));
18       }
19       index++;
20    }
21    return subset;
22 }
```

There's nothing substantially better or worse about this solution compared to the first one.

**9.5** *Write a method to compute all permutations of a string*

*pg 109*

**SOLUTION**

Like in many recursive problems, the Base Case and Build approach will be useful. Assume we have a string S represented by the characters $a_1a_2...a_n$.

*Base Case:* `n = 1`

The only permutation of `S = a1` is the string $a_1$.

*Case:* `n = 2`

The permutations of `S = a1a2` are the strings $a_1a_2$ and $a_2a_1$.

*Case:* `n = 3`

Here is where the cases get more interesting. How can we generate all permutations of $a_1a_2a_3$ given the permutations of $a_1a_2$? That is, we need to generate

$$a_1a_2a_3,\ a_1a_3a_2,\ a_2a_1a_3,\ a_2a_3a_1,\ a_3a_1a_2,\ a_3a_2a_1$$

given

$$a_1a_2,\ a_2a_1.$$

The difference between these lists is that the first one contains $a_3$ while the second list does not. So, how can we generate `f(3)` from `f(2)`? By pushing $a_3$ into every possible spot in the strings in `f(2)`.

*Case:* `n > 0`

For the general case, we just repeat this process. We solve for `f(n-1)`, and then push $a_n$ into every spot in each of these strings.

The code below does just this.

```
1  public static ArrayList<String> getPerms(String str) {
2     if (str == null) {
```

```
3        return null;
4     }
5     ArrayList<String> permutations = new ArrayList<String>();
6     if (str.length() == 0) { // base case
7        permutations.add("");
8        return permutations;
9     }
10
11    char first = str.charAt(0); // get the first character
12    String remainder = str.substring(1); // remove the 1st character
13    ArrayList<String> words = getPerms(remainder);
14    for (String word : words) {
15       for (int j = 0; j <= word.length(); j++) {
16          String s = insertCharAt(word, first, j);
17          permutations.add(s);
18       }
19    }
20    return permutations;
21 }
22
23 public static String insertCharAt(String word, char c, int i) {
24    String start = word.substring(0, i);
25    String end = word.substring(i);
26    return start + c + end;
27 }
```

This solution takes O(n!) time, since there are n! permutations. We cannot do better than this.

**9.6** *Implement an algorithm to print all valid (i.e., properly opened and closed) combinations of n-pairs of parentheses.*

*pg 110*

**SOLUTION**

Our first thought here might be to apply a recursive approach where we build the solution for f(n) by adding pairs of parentheses to f(n-1). That's certainly a good instinct.

Let's consider the solution for n = 3:

```
(()())     ((()))     ()(())     (())()     ()()()
```

How might we build this from n = 2?

```
(())     ()()
```

We can do this by inserting a pair of parentheses inside every existing pair of parentheses, as well as one at the beginning of the string. Any other places that we could insert parentheses, such as at the end of the string, would reduce to the earlier cases.

So, we have the following:

```
(()) -> (()()) /* inserted pair after 1st left paren */
     -> ((())) /* inserted pair after 2nd left paren */
     -> ()(()) /* inserted pair at beginning of string */
()() -> (())() /* inserted pair after 1st left paren */
     -> ()(()) /* inserted pair after 2nd left paren */
     -> ()()() /* inserted pair at beginning of string */
```

But wait—we have some duplicate pairs listed. The string `()(())` is listed twice.

If we're going to apply this approach, we'll need to check for duplicate values before adding a string to our list.

```
1  public static Set<String> generateParens(int remaining) {
2    Set<String> set = new HashSet<String>();
3    if (remaining == 0) {
4      set.add("");
5    } else {
6      Set<String> prev = generateParens(remaining - 1);
7      for (String str : prev) {
8        for (int i = 0; i < str.length(); i++) {
9          if (str.charAt(i) == '(') {
10           String s = insertInside(str, i);
11           /* Add s to set if it's not already in there. Note:
12            * HashSet automatically checks for duplicates before
13            * adding, so an explicit check is not necessary. */
14           set.add(s);
15         }
16       }
17       if (!set.contains("()" + str)) {
18         set.add("()" + str);
19       }
20     }
21   }
22   return set;
23 }
24
25 public String insertInside(String str, int leftIndex) {
26   String left = str.substring(0, leftIndex + 1);
27   String right = str.substring(leftIndex + 1, str.length());
28   return left + "()" + right;
29 }
```

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.

```
1  public void addParen(ArrayList<String> list, int leftRem,
2                       int rightRem, char[] str, int count) {
3     if (leftRem < 0 || rightRem < leftRem) return; // invalid state
4
5     if (leftRem == 0 && rightRem == 0) { /* no more parens left */
6        String s = String.copyValueOf(str);
7        list.add(s);
8     } else {
9        /* Add left paren, if there are any left parens remaining. */
10       if (leftRem > 0) {
11          str[count] = '(';
12          addParen(list, leftRem - 1, rightRem, str, count + 1);
13       }
14
15       /* Add right paren, if expression is valid */
16       if (rightRem > leftRem) {
17          str[count] = ')';
18          addParen(list, leftRem, rightRem - 1, str, count + 1);
19       }
20    }
21 }
22
23 public ArrayList<String> generateParens(int count) {
24    char[] str = new char[count*2];
25    ArrayList<String> list = new ArrayList<String>();
26    addParen(list, count, count, str, 0);
27    return list;
28 }
```

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.

**9.7** *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 110*

### 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 {
2    Black, White, Red, Yellow, Green
3  }
4
5  boolean paintFill(Color[][] screen, int x, int y, Color ocolor,
6                    Color ncolor) {
7    if (x < 0 || x >= screen[0].length ||
8        y < 0 || y >= screen.length) {
9      return false;
10   }
11   if (screen[y][x] == ocolor) {
12     screen[y][x] = ncolor;
13     paintFill(screen, x - 1, y, ocolor, ncolor); // left
14     paintFill(screen, x + 1, y, ocolor, ncolor); // right
15     paintFill(screen, x, y - 1, ocolor, ncolor); // top
16     paintFill(screen, x, y + 1, ocolor, ncolor); // bottom
17   }
18   return true;
19 }
20
21 boolean paintFill(Color[][] screen, int x, int y, Color ncolor) {
22   if (screen[y][x] == ncolor) return false;
23   return paintFill(screen, x, y, screen[y][x], ncolor);
24 }
```

Note the ordering of the x and y in `screen[y][x]`, and remember this when you hit graphical problem. Because x represents the *horizontal* axis (that is, it's left to right), it actually corresponds to the number of a column, not the number of rows. 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.

**9.8** *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 110*

### SOLUTION

This is a recursive problem, so let's figure out how to compute `makeChange(n)` using

prior solutions (i.e., sub-problems).

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 sub-problems?

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  public int makeChange(int n, int denom) {
2     int next_denom = 0;
3     switch (denom) {
4     case 25:
5        next_denom = 10;
6        break;
7     case 10:
8        next_denom = 5;
9        break;
10    case 5:
11       next_denom = 1;
12       break;
13    case 1:
14       return 1;
15    }
16
17    int ways = 0;
18    for (int i = 0; i * denom <= n; i++) {
19       ways += makeChange(n - i * denom, next_denom);
20    }
21    return ways;
22 }
23
24 System.out.writeln(makeChange(100, 25));
```

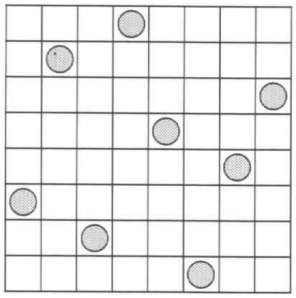
Although we've implemented this to work for US currency, it can be easily extended to work for any other set of denominations.

**9.9** *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 110*

## 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[row] = c` indicates that row `r` has a queen at column `c`.

**9.10** *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 build the tallest stack possible, where the height of a stack is the sum of the heights of each box.*

*pg 110*

**SOLUTION**

To tackle this problem, we need to recognize the relationship between the different sub-problems.

Imagine we had the following boxes: $b_1$, $b_2$, ..., $b_n$. 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 $b_5$.

The code below implements this algorithm recursively.

```
1  public ArrayList<Box> createStackR(Box[] boxes, Box bottom) {
2     int max_height = 0;
3     ArrayList<Box> max_stack = null;
4     for (int i = 0; i < boxes.length; i++) {
5        if (boxes[i].canBeAbove(bottom)) {
6           ArrayList<Box> new_stack = createStackR(boxes, boxes[i]);
7           int new_height = stackHeight(new_stack);
8           if (new_height > max_height) {
9              max_stack = new_stack;
10             max_height = new_height;
11          }
12       }
13    }
14
15    if (max_stack == null) {
16       max_stack = new ArrayList<Box>();
17    }
```

```
18    if (bottom != null) {
19       max_stack.add(0, bottom); // Insert in bottom of stack
20    }
21
22    return max_stack;
23 }
```

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 dynamic programming.

```
1  public ArrayList<Box> createStackDP(Box[] boxes, Box bottom,
2        HashMap<Box, ArrayList<Box>> stack_map) {
3     if (bottom != null && stack_map.containsKey(bottom)) {
4        return stack_map.get(bottom);
5     }
6
7     int max_height = 0;
8     ArrayList<Box> max_stack = null;
9     for (int i = 0; i < boxes.length; i++) {
10       if (boxes[i].canBeAbove(bottom)) {
11          ArrayList<Box> new_stack =
12             createStackDP(boxes, boxes[i], stack_map);
13          int new_height = stackHeight(new_stack);
14          if (new_height > max_height) {
15             max_stack = new_stack;
16             max_height = new_height;
17          }
18       }
19    }
20
21    if (max_stack == null) max_stack = new ArrayList<Box>();
22    if (bottom != null) max_stack.add(0, bottom);
23    stack_map.put(bottom, max_stack);
24
25    return (ArrayList<Box>)max_stack.clone();
26 }
```

You might ask why, in line 25, we have to cast `max_stack.clone()`. Isn't `max_stack` already of the correct data type? Yes, but we still need to cast.

The `clone()` method originally comes from the `Object` class and has the signature:

```
1  protected Object clone() { ... }
```

When we override a method, we can change the parameters, but we cannot change the return type. Therefore, if Foo were to inherit from `Object` and override `clone()`, its `clone()` method will still return an instance of type `Object`.

This is precisely what happens with the statement `(ArrayList<Box>)max_stack.clone()`. The stack class overrides `clone()`, but the method still must return an

`Object`. So, we must cast its return value.

**9.11** *Given a boolean expression consisting of the symbols `0`, `1`, `&`, `|`, and `^`, 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`.*

*pg 110*

**SOLUTION**

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

Suppose `int f(expression, result)` is a function which returns the count of all valid expressions which evaluate to result. We want to compute `f(1^0|0|1, true)` (that is, all ways of parenthesizing the expression `1^0|0|1` such that the expression evaluates to `true`). Each parenthesized expression must have an outermost pair of parentheses. So, we can say that:

```
f(1^0|0|1, true) = f(1 ^ (0|0|1), true) +
                   f((1^0) | (0|1), true) +
                   f((1^0|0) | 1, true)
```

That is, we can iterate through the expression, treating each operator as the first operator to be parenthesized.

Now, how do we compute one of these inner expressions, like `f((1^0) | (0|1), true)`? Well, in order for that expression to evaluate to `true`, either the left half or the right half must evaluate to `true`. So, the expression breaks down as:

```
f((1^0) | (0|1), true) =   f(1^0, true)  * f(0|1, true) +
                           f(1^0, false) * f(0|1, true) +
                           f(1^0, true)  * f(0|1, false)
```

We can implement a similar break down for each of the boolean operators:

```
f(exp1 | exp2, true)   =   f(exp1, true)  * f(exp2, true) +
                           f(exp1, true)  * f(exp2, false) +
                           f(exp1, false) * f(exp2, true)
f(exp1 & exp2, true)   =   f(exp1, true)  * f(exp2, true)
f(exp1 ^ exp2, true)   =   f(exp1, true)  * f(exp2, false) +
                           f(exp1, false) * f(exp2, true)
```

For the `false` results, we can perform a very similar operation:

```
f(exp1 | exp2, false)  =   f(exp1, false) * f(exp2, false)
f(exp1 & exp2, false)  =   f(exp1, false) * f(exp2, false) +
                           f(exp1, true)  * f(exp2, false) +
                           f(exp1, false) * f(exp2, true)
f(exp1 ^ exp2, false)  =   f(exp1, true)  * f(exp2, true) +
                           f(exp1, false) * f(exp2, false)
```

Implementing this is now just a matter of applying this recurrence relation. (Note: to

keep the lines from wrapping unnecessarily and making the code very confusing, we've implemented this with extra short variable names.)

```
1  public int f(String exp, boolean result, int s, int e) {
2     if (s == e) {
3        if (exp.charAt(s) == '1' && result) {
4           return 1;
5        } else if (exp.charAt(s) == '0' && !result) {
6           return 1;
7        }
8        return 0;
9     }
10    int c = 0;
11    if (result) {
12       for (int i = s + 1; i <= e; i += 2) {
13          char op = exp.charAt(i);
14          if (op == '&') {
15             c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
16          } else if (op == '|') {
17             c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
18             c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
19             c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
20          } else if (op == '^') {
21             c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
22             c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
23          }
24       }
25    } else {
26       for (int i = s + 1; i <= e; i += 2) {
27          char op = exp.charAt(i);
28          if (op == '&') {
29             c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
30             c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
31             c += f(exp, false, s, i - 1) * f(exp, false, i + 1,e);
32          } else if (op == '|') {
33             c += f(exp, false, s, i - 1) * f(exp, false, i + 1,e);
34          } else if (op == '^') {
35             c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
36             c += f(exp, false, s, i - 1) * f(exp, false, i + 1,e);
37          }
38       }
39    }
40    return c;
41 }
```

Although this works, it's not very efficient. This method will recompute `f(exp)` many times, for the same value of exp.

To solve this issue, we can use dynamic programming and cache the results of the different expressions. Note that we need to cache based on the `expression` and the `result`.

```
1  public int f(String exp, boolean result, int s, int e,
2               HashMap<String, Integer> q) {
3     String key = "" + result + s + e;
4     if (q.containsKey(key)) {
5        return q.get(key);
6     }
7
8     if (s == e) {
9        if (exp.charAt(s) == '1' && result == true) {
10          return 1;
11       } else if (exp.charAt(s) == '0' && result == false) {
12          return 1;
13       }
14       return 0;
15    }
16    int c = 0;
17    if (result) {
18       for (int i = s + 1; i <= e; i += 2) {
19          char op = exp.charAt(i);
20          if (op == '&') {
21             c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
22          } else if (op == '|') {
23             c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
24             c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
25             c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
26          } else if (op == '^') {
27             c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
28             c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
29          }
30       }
31    } else {
32       for (int i = s + 1; i <= e; i += 2) {
33          char op = exp.charAt(i);
34          if (op == '&') {
35             c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
36             c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
37             c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
38          } else if (op == '|') {
39             c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
40          } else if (op == '^') {
41             c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
42             c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
43          }
44       }
45    }
46    q.put(key, c);
47    return c;
48 }
```

While this is nicely optimized with dynamic programming, it's still not as optimized as it

could be. If we knew how many ways there were of parenthesizing an expression, then we could compute `f(exp = false)` by doing `total(exp) - f(exp = true)`.

There *is* a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where n is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

With this adjustment, the code looks like the following.

```
1  public int f(String exp, boolean result, int s, int e,
2          HashMap<String, Integer> q) {
3     String key = "" + s + e;
4     int c = 0;
5     if (!q.containsKey(key)) {
6        if (s == e) {
7           if (exp.charAt(s) == '1') c = 1;
8           else c = 0;
9        }
10
11       for (int i = s + 1; i <= e; i += 2) {
12          char op = exp.charAt(i);
13          if (op == '&') {
14             c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
15          } else if (op == '|') {
16             int left_ops = (i-1-s)/2; // parens on left
17             int right_ops = (e - i - 1) / 2;  // parens on right
18             int total_ways = total(left_ops) * total(right_ops);
19             int total_false = f(exp,false,s,i-1,q) *
20                               f(exp,false,i+1,e,q);
21             c += total_ways - total_false;
22          } else if (op == '^') {
23             c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
24             c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
25          }
26       }
27       q.put(key, c);
28    } else {
29       c = q.get(key);
30    }
31    if (result) {
32       return c;
33    } else {
34       int num_ops = (e - s) / 2;
35       return total(num_ops) - c;
36    }
37 }
```