```
1     // Return minimum coins to make change
2     // Simple recursive algorithm that is very inefficient
3
4     public static int makeChange( int [ ] coins, int change,
5                                   int differentCoins )
6     {
7         int minCoins = change;
8
9             // Look for exact match with any single coin
10        for( int i = 0; i < differentCoins; i++ )
11            if( coins[ i ] == change )
12                return 1;
13
14            // No match; solve recursively
15        for( int j = 1; j <= change / 2; j++ )
16        {
17            int thisCoins =
18              makeChange( coins, j, differentCoins ) +
19              makeChange( coins, change - j, differentCoins );
20            if( thisCoins < minCoins )
21                minCoins = thisCoins;
22        }
23
24        return minCoins;
25    }
```

Figure 7.21 Inefficient recursive method to solve the change-making problem

An alternative recursive change-making algorithm is still inefficient.

An alternative algorithm is to reduce the problem recursively by specifying one of the coins. For example, for 63 cents, we can give change in the following ways shown in Figure 7.22:

- One 1-cent piece plus 62 cents recursively distributed
- One 5-cent piece plus 58 cents recursively distributed
- One 10-cent piece plus 53 cents recursively distributed
- One 21-cent piece plus 42 cents recursively distributed
- One 25-cent piece plus 38 cents recursively distributed

Instead of solving 62 recursive problems, as was done in Figure 7.20, we get by with only five recursive calls, one for each different coin. Once again, a naive recursive implementation would be very inefficient because it would recompute answers. For example, in the first case we are left with a problem of making 62 cents in change. In this subproblem, one of the recursive calls that is made chooses a 10-cent piece and recursively solves for 52 cents. In the third case, we are left with 53 cents. One of its recursive calls removes the1-cent piece and also recursively solves for 52 cents. This redundant work again leads to a wildly large running time. If we are careful, however, we can make the algorithm reasonably fast.