```
10
11 ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
12    ArrayList<Integer> subset = new ArrayList<Integer>();
13    int index = 0;
14    for (int k = x; k > 0; k >>= 1) {
15       if ((k & 1) == 1) {
16          subset.add(set.get(index));
17       }
18       index++;
19    }
20    return subset;
21 }
```

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

8.5 **Recursive Multiply:** Write a recursive function to multiply two positive integers without using the * operator (or / operator). You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

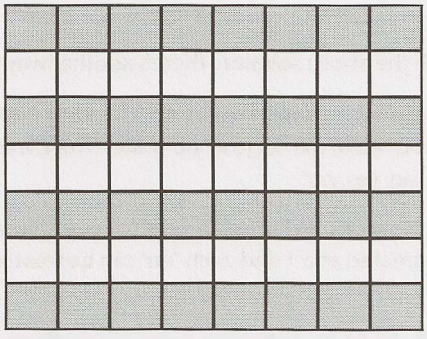
pg 135

SOLUTION

Let's pause for a moment and think about what it means to do multiplication.

> This is a good approach for a lot of interview questions. It's often useful to think about what it really means to do something, even when it's pretty obvious.

We can think about multiplying 8x7 as doing 8+8+8+8+8+8+8 (or adding 7 eight times). We can also think about it as the number of squares in an 8x7 grid.



Solution #1

How would we count the number of squares in this grid? We could just count each cell. That's pretty slow, though.

Alternatively, we could count half the squares and then double it (by adding this count to itself). To count half the squares, we repeat the same process.

Of course, this "doubling" only works if the number is in fact even. When it's not even, we need to do the counting/summing from scratch.

```
1  int minProduct(int a, int b) {
2     int bigger = a < b ? b : a;
```

```
  int smaller = a < b ? a : b;
  return minProductHelper(smaller, bigger);
}

int minProductHelper(int smaller, int bigger) {
  if (smaller == 0) { // 0 x bigger = 0
    return 0;
  } else if (smaller == 1) { // 1 x bigger = bigger
    return bigger;
  }

  /* Compute half. If uneven, compute other half. If even, double it. */
  int s = smaller >> 1; // Divide by 2
  int side1 = minProduct(s, bigger);
  int side2 = side1;
  if (smaller % 2 == 1) {
    side2 = minProductHelper(smaller - s, bigger);
  }

  return side1 + side2;
}
```

Can we do better? Yes.

Solution #2

If we observe how the recursion operates, we'll notice that we have duplicated work. Consider this example:

```
        minProduct(17, 23)
             minProduct(8, 23)
                 minProduct(4, 23) * 2
                     ...
           + minProduct(9, 23)
                 minProduct(4, 23)
                     ...
             + minProduct(5, 23)
                     ...
```

The second call to `minProduct(4, 23)` is unaware of the prior call, and so it repeats the same work. We should cache these results.

```
int minProduct(int a, int b) {
  int bigger = a < b ? b : a;
  int smaller = a < b ? a : b;

  int memo[] = new int[smaller + 1];
  return minProduct(smaller, bigger, memo);
}

int minProduct(int smaller, int bigger, int[] memo) {
  if (smaller == 0) {
    return 0;
  } else if (smaller == 1) {
    return bigger;
  } else if (memo[smaller] > 0) {
    return memo[smaller];
  }

  /* Compute half. If uneven, compute other half. If even, double it. */
```

```
    int s = smaller >> 1; // Divide by 2
    int side1 = minProduct(s, bigger, memo); // Compute half
    int side2 = side1;
    if (smaller % 2 == 1) {
       side2 = minProduct(smaller - s, bigger, memo);
    }

    /* Sum and cache.*/
    memo[smaller] = side1 + side2;
    return memo[smaller];
}
```

We can still make this a bit faster.

Solution #3

One thing we might notice when we look at this code is that a call to `minProduct` on an even number is much faster than one on an odd number. For example, if we call `minProduct(30, 35)`, then we'll just do `minProduct(15, 35)` and double the result. However, if we do `minProduct(31, 35)`, then we'll need to call `minProduct(15, 35)` and `minProduct(16, 35)`.

This is unnecessary. Instead, we can do:

```
minProduct(31, 35) = 2 * minProduct(15, 35) + 35
```

After all, since `31 = 2*15+1`, then `31x35 = 2*15*35+35`.

The logic in this final solution is that, on even numbers, we just divide `smaller` by 2 and double the result of the recursive call. On odd numbers, we do the same, but then we also add `bigger` to this result.

In doing so, we have an unexpected "win." Our `minProduct` function just recurses straight downwards, with increasingly small numbers each time. It will never repeat the same call, so there's no need to cache any information.

```
int minProduct(int a, int b) {
   int bigger = a < b ? b : a;
   int smaller = a < b ? a : b;
   return minProductHelper(smaller, bigger);
}

int minProductHelper(int smaller, int bigger) {
   if (smaller == 0) return 0;
   else if (smaller == 1) return bigger;

   int s = smaller >> 1; // Divide by 2
   int halfProd = minProductHelper(s, bigger);

   if (smaller % 2 == 0) {
      return halfProd + halfProd;
   } else {
      return halfProd + halfProd + bigger;
   }
}
```

This algorithm will run in `O(log s)` time, where `s` is the smaller of the two numbers.

8.6 **Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

(1) Only one disk can be moved at a time.

(2) A disk is slid off the top of one tower onto another tower.

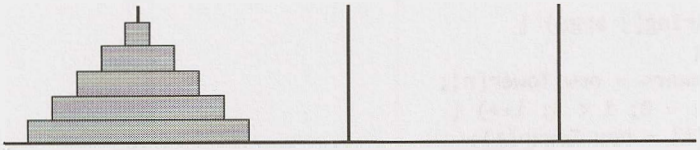
(3) A disk cannot be placed on top of a smaller disk.

Write a program to move the disks from the first tower to the last using Stacks.

pg 135

SOLUTION

This problem sounds like a good candidate for the Base Case and Build approach.



Let's start with the smallest possible example: n = 1.

Case n = 1. Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case n = 2. Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move other disks to Tower 3.

Case n = 3. Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—just repeat what we did in Step 1.

Case n = 4. Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

```
moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
  /* Base case */
  if (n <= 0) return;

  /* move top n - 1 disks from origin to buffer, using destination as a buffer. */
  moveDisks(n - 1, origin, buffer, destination);

  /* move top from origin to destination
  moveTop(origin, destination);

  /* move top n - 1 disks from buffer to destination, using origin as a buffer. */
  moveDisks(n - 1, buffer, destination, origin);
}
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
void main(String[] args) {
  int n = 3;
  Tower[] towers = new Tower[n];
  for (int i = 0; i < 3; i++) {
    towers[i] = new Tower(i);
  }

  for (int i = n - 1; i >= 0; i--) {
    towers[0].add(i);
  }
  towers[0].moveDisks(n, towers[2], towers[1]);
}

class Tower {
  private Stack<Integer> disks;
  private int index;
  public Tower(int i) {
    disks = new Stack<Integer>();
    index = i;
  }

  public int index() {
    return index;
  }

  public void add(int d) {
    if (!disks.isEmpty() && disks.peek() <= d) {
      System.out.println("Error placing disk " + d);
    } else {
      disks.push(d);
    }
  }

  public void moveTopTo(Tower t) {
    int top = disks.pop();
    t.add(top);
  }

```

```
39     public void moveDisks(int n, Tower destination, Tower buffer) {
40       if (n > 0) {
41         moveDisks(n - 1, buffer, destination);
42         moveTopTo(destination);
43         buffer.moveDisks(n - 1, destination, this);
44       }
45     }
46   }
```

Implementing the towers as their own objects is not strictly necessary, but it does help to make the code cleaner in some respects.

8.7 Permutations without Dups: Write a method to compute all permutations of a string of unique characters.

pg 135

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 $a_1a_2...a_n$.

Approach 1: Building from permutations of first n-1 characters.

Base Case: permutations of first character substring

The only permutation of a_1 is the string a_1. So:

$P(a_1) = a_1$

Case: permutations of a_1a_2

$P(a_1a_2) = a_1a_2$ and a_2a_1

Case: permutations of $a_1a_2a_3$

$P(a_1a_2a_3) = a_1a_2a_3,\ a_1a_3a_2,\ a_2a_1a_3,\ a_2a_3a_1,\ a_3a_1a_2,\ a_3a_2a_1,$

Case: permutations of $a_1a_2a_3a_4$

This is the first interesting case. How can we generate permutations of $a_1a_2a_3a_4$ from $a_1a_2a_3$?

Each permutation of $a_1a_2a_3a_4$ represents an ordering of $a_1a_2a_3$. For example, $a_2a_4a_1a_3$ represents the order $a_2a_1a_3$.

Therefore, if we took all the permutations of $a_1a_2a_3$ and added a_4 into all possible locations, we would get all permutations of $a_1a_2a_3a_4$.

$a_1a_2a_3$ -> $a_4a_1a_2a_3,\ a_1a_4a_2a_3,\ a_1a_2a_4a_3,\ a_1a_2a_3a_4$
$a_1a_3a_2$ -> $a_4a_1a_3a_2,\ a_1a_4a_3a_2,\ a_1a_3a_4a_2,\ a_1a_3a_2a_4$
$a_3a_1a_2$ -> $a_4a_3a_1a_2,\ a_3a_4a_1a_2,\ a_3a_1a_4a_2,\ a_3a_1a_2a_4$
$a_2a_1a_3$ -> $a_4a_2a_1a_3,\ a_2a_4a_1a_3,\ a_2a_1a_4a_3,\ a_2a_1a_3a_4$
$a_2a_3a_1$ -> $a_4a_2a_3a_1,\ a_2a_4a_3a_1,\ a_2a_3a_4a_1,\ a_2a_3a_1a_4$
$a_3a_2a_1$ -> $a_4a_3a_2a_1,\ a_3a_4a_2a_1,\ a_3a_2a_4a_1,\ a_3a_2a_1a_4$

We can now implement this algorithm recursively.

```
1  ArrayList<String> getPerms(String str) {
2    if (str == null) return null;
3
4    ArrayList<String> permutations = new ArrayList<String>();
5    if (str.length() == 0) { // base case
6      permutations.add("");
```

```
      return permutations;
    }

    char first = str.charAt(0); // get the first char
    String remainder = str.substring(1); // remove the first char
    ArrayList<String> words = getPerms(remainder);
    for (String word : words) {
      for (int j = 0; j <= word.length(); j++) {
        String s = insertCharAt(word, first, j);
        permutations.add(s);
      }
    }
    return permutations;
  }

  /* Insert char c at index i in word. */
  String insertCharAt(String word, char c, int i) {
    String start = word.substring(0, i);
    String end = word.substring(i);
    return start + c + end;
  }
```

Approach 2: Building from permutations of all n-1 character substrings.

Base Case: single-character strings

The only permutation of a_1 is the string a_1. So:

$P(a_1) = a_1$

Case: two-character strings

$P(a_1a_2) = a_1a_2$ and a_2a_1.
$P(a_2a_3) = a_2a_3$ and a_3a_2.
$P(a_1a_3) = a_1a_3$ and a_3a_1.

Case: three-character strings

Here is where the cases get more interesting. How can we generate all permutations of three-character strings, such as $a_1a_2a_3$, given the permutations of two-character strings?

Well, in essence, we just need to "try" each character as the first character and then append the permutations.

$$P(a_1a_2a_3) = \{a_1 + P(a_2a_3)\} + a_2 + P(a_1a_3)\} + \{a_3 + P(a_1a_2)\}$$
$$\{a_1 + P(a_2a_3)\} \rightarrow a_1a_2a_3,\ a_1a_3a_2$$
$$\{a_2 + P(a_1a_3)\} \rightarrow a_2a_1a_3,\ a_2a_3a_1$$
$$\{a_3 + P(a_1a_2)\} \rightarrow a_3a_1a_2,\ a_3a_2a_1$$

Now that we can generate all permutations of three-character strings, we can use this to generate permutations of four-character strings.

$$P(a_1a_2a_3a_4) = \{a_1 + P(a_2a_3a_4)\} + \{a_2 + P(a_1a_3a_4)\} + \{a_3 + P(a_1a_2a_4)\} + \{a_4 + P(a_1a_2a_3)\}$$

This is now a fairly straightforward algorithm to implement.

```
1  ArrayList<String> getPerms(String remainder) {
2    int len = remainder.length();
3    ArrayList<String> result = new ArrayList<String>();
4
5    /* Base case. */
```

```
6     if (len == 0) {
7       result.add(""); // Be sure to return empty string!
8       return result;
9     }
10
11
12    for (int i = 0; i < len; i++) {
13      /* Remove char i and find permutations of remaining chars.*/
14      String before = remainder.substring(0, i);
15      String after = remainder.substring(i + 1, len);
16      ArrayList<String> partials = getPerms(before + after);
17
18      /* Prepend char i to each permutation.*/
19      for (String s : partials) {
20        result.add(remainder.charAt(i) + s);
21      }
22    }
23
24    return result;
25  }
```

Alternatively, instead of passing the permutations back up the stack, we can push the prefix down the stack. When we get to the bottom (base case), `prefix` holds a full permutation.

```
1   ArrayList<String> getPerms(String str) {
2     ArrayList<String> result = new ArrayList<String>();
3     getPerms("", str, result);
4     return result;
5   }
6
7   void getPerms(String prefix, String remainder, ArrayList<String> result) {
8     if (remainder.length() == 0) result.add(prefix);
9
10    int len = remainder.length();
11    for (int i = 0; i < len; i++) {
12      String before = remainder.substring(0, i);
13      String after = remainder.substring(i + 1, len);
14      char c = remainder.charAt(i);
15      getPerms(prefix + c, before + after, result);
16    }
17  }
```

For a discussion of the runtime of this algorithm, see Example 12 on page 51.

8.8 Permutations with Duplicates: Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

pg 135

SOLUTION

This is very similar to the previous problem, except that now we could potentially have duplicate characters in the word.

One simple way of handling this problem is to do the same work to check if a permutation has been created before and then, if not, add it to the list. A simple hash table will do the trick here. This solution will take $O(n!)$ time in the worst case (and, in fact, in all cases).

While it's true that we can't beat this worst case time, we should be able to design an algorithm to beat this in many cases. Consider a string with all duplicate characters, like aaaaaaaaaaaaa. This will take an extremely long time (since there are over 6 billion permutations of a 13-character string), even though there is only one unique permutation.

Ideally, we would like to only create the unique permutations, rather than creating every permutation and then ruling out the duplicates.

We can start with computing the count of each letter (easy enough to get this—just use a hash table). For a string such as aabbbbc, this would be:

```
a->2 | b->4 | c->1
```

Let's imagine generating a permutation of this string (now represented as a hash table). The first choice we make is whether to use an a, b, or c as the first character. After that, we have a subproblem to solve: find all permutations of the remaining characters, and append those to the already picked "prefix."

```
P(a->2 | b->4 | c->1) = {a + P(a->1 | b->4 | c->1)} +
                        {b + P(a->2 | b->3 | c->1)} +
                        {c + P(a->2 | b->4 | c->0)}
    P(a->1 | b->4 | c->1) = {a + P(a->0 | b->4 | c->1)} +
                            {b + P(a->1 | b->3 | c->1)} +
                            {c + P(a->1 | b->4 | c->0)}
    P(a->2 | b->3 | c->1) = {a + P(a->1 | b->3 | c->1)} +
                            {b + P(a->2 | b->2 | c->1)} +
                            {c + P(a->2 | b->3 | c->0)}
    P(a->2 | b->4 | c->0) = {a + P(a->1 | b->4 | c->0)} +
                            {b + P(a->2 | b->3 | c->0)}
```

Eventually, we'll get down to no more characters remaining.

The code below implements this algorithm.

```
1  ArrayList<String> printPerms(String s) {
2    ArrayList<String> result = new ArrayList<String>();
3    HashMap<Character, Integer> map = buildFreqTable(s);
4    printPerms(map, "", s.length(), result);
5    return result;
6  }
7
8  HashMap<Character, Integer> buildFreqTable(String s) {
9    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
10   for (char c : s.toCharArray()) {
11     if (!map.containsKey(c)) {
12       map.put(c, 0);
13     }
14     map.put(c, map.get(c) + 1);
15   }
16   return map;
17 }
18
19 void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,
20                 ArrayList<String> result) {
21   /* Base case. Permutation has been completed. */
22   if (remaining == 0) {
23     result.add(prefix);
24     return;
25   }
26
27   /* Try remaining letters for next char, and generate remaining permutations. */
```

```
28    for (Character c : map.keySet()) {
29      int count = map.get(c);
30      if (count > 0) {
31        map.put(c,  count - 1);
32        printPerms(map, prefix + c, remaining - 1, result);
33        map.put(c,  count);
34      }
35    }
36  }
```

In situations where the string has many duplicates, this algorithm will run a lot faster than the earlier algorithm.

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

EXAMPLE

Input: 3

Output: ((())), (()()), (())(), ()(()), ()()()

pg 136

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  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++) {
```