**Exercise 12.19: countBinary**

**Question**

Write a method countBinary that accepts an integer n as a parameter and that prints all binary numbers that have n digits in ascending order, printing each value on a separate line. All n digits should be shown for all numbers, including leading zeros if necessary. You may assume that n is non-negative. If n is 0, a blank line of output should be produced. Do not use a loop in your solution; implement it recursively.

|  |  |
| --- | --- |
| **Call** | **Output** |
| countBinary(1); | 0  1 |
| countBinary(2); | 00  01  10  11 |
| countBinary(3); | 000  001  010  011  100  101  110  111 |

Hint: It may help to define a private helper method that accepts different parameters than the original method. In particular, consider building up a set of characters as a String for eventual printing.

**Solution:**

public static void countBinary(int n) {

countBinary(n, "");

}

private static void countBinary(int digitsLeft, String s) {

if (digitsLeft == 0) {

System.out.println(s);

} else {

countBinary(digitsLeft - 1, s + "0");

countBinary(digitsLeft - 1, s + "1");

}

}

## Exercise 12.20: subsets

**Question**

Write a method subsets that uses recursive backtracking to find every possible sub-list of a given list. A sub-list of a list L contains 0 or more of L's elements. Your method should accept a List of strings as its parameter and print every sub-list that could be created from elements of that list, one per line. For example, suppose a variable called list stores the following elements:

[Janet, Robert, Morgan, Char]

The call of subsets(list); would produce output such as the following:

[Janet, Robert, Morgan, Char]

[Janet, Robert, Morgan]

[Janet, Robert, Char]

[Janet, Robert]

[Janet, Morgan, Char]

[Janet, Morgan]

[Janet, Char]

[Janet]

[Robert, Morgan, Char]

[Robert, Morgan]

[Robert, Char]

[Robert]

[Morgan, Char]

[Morgan]

[Char]

[]

The order in which you show the sub-lists does not matter, and the order of the elements of each sub-list also does not matter. The key thing is that your method should produce the correct overall set of sub-lists as its output. Notice that the empty list is considered one of these sub-lists. You may assume that the list passed to your method is not null and that the list contains no duplicates. Do not use any loops in solving this problem.

Hint: This problem is somewhat similar to the permutations problem. Consider each element and try to generate all sub-lists that do include it, as well as all sub-lists that do not include it.

It can be hard to see a pattern from looking at the lines of output. But notice that the first 8 of 16 total lines of output constitute all the sets that include Janet, and the last 8 lines are the sets that do not have her as a member. Within either of those groups of 8 lines, the first 4 of them are all the sets that include Robert, and the last 4 lines are the ones that do not include him. Within a clump of 4, the first 2 are the ones including Morgan, and the last 2 are the ones that do not include Morgan. And so on. Once again, you do not have to match this exact order, but looking at it can help with figuring out the patterns and the recursion.

**Solution:**

// Prints all sub-lists of the given list of Strings.

// Precondition: elements != null and elements contains no duplicates

public static void subsets(List<String> elements) {

List<String> chosen = new ArrayList<String>();

explore(elements, chosen);

}

// Private recursive helper to explore all sub-lists of the given list of

// elements, assuming the given list of strings have already been chosen.

private static void explore(List<String> elements, List<String> chosen) {

if (elements.isEmpty()) {

System.out.println(chosen); // base case; nothing left to choose

} else {

String first = elements.remove(0); // make a choice: 1st element

// two explorations: one with this first element, one without

chosen.add(first);

explore(elements, chosen);

chosen.remove(chosen.size() - 1);

explore(elements, chosen);

elements.add(0, first); // backtrack! put 1st element back

}

}

## Exercise 12.21: maxSum

**Question**

Write a recursive method maxSum that accepts a list of integers L and an integer limit n as its parameters and uses backtracking to find the maximum sum that can be generated by adding elements of L that does not exceed n. For example, if you are given the list of integers [7, 30, 8, 22, 6, 1, 14] and the limit of 19, the maximum sum that can be generated that does not exceed is 16, achieved by adding 7, 8, and 1. If the list L is empty, or if the limit is not a positive integer, or all of L's values exceed the limit, return 0.

Each index's element in the list can be added to the sum only once, but the same number value might occur more than once in a list, in which case each occurrence might be added to the sum. For example, if the list is [6, 2, 1] you may use up to one 6 in the sum, but if the list is [6, 2, 6, 1] you may use up to two sixes.

Here are several example calls to your method and their expected return values:

|  |  |  |
| --- | --- | --- |
| **List L** | **Limit n** | **maxSum(L, n) returns** |
| [7, 30, 8, 22, 6, 1, 14] | 19 | 16 |
| [5, 30, 15, 13, 8] | 42 | 41 |
| [30, 15, 20] | 40 | 35 |
| [6, 2, 6, 9, 1] | 30 | 24 |
| [11, 5, 3, 7, 2] | 14 | 14 |
| [10, 20, 30] | 7 | 0 |
| [10, 20, 30] | 20 | 20 |
| [] | 10 | 0 |

You may assume that all values in the list are non-negative. Your method may alter the contents of the list L as it executes, but L should be restored to its original state before your method returns. Do not use any loops in solving this problem.

**Solution:**

public static int maxSum(List<Integer> numbers, int limit) {

if (limit <= 0 || numbers.isEmpty()) {

return 0;

} else {

int first = numbers.get(0);

numbers.remove(0);

int max;

if (first > limit) {

max = maxSum(numbers, limit);

} else {

int with = first + maxSum(numbers, limit - first);

int without = maxSum(numbers, limit);

max = Math.max(with, without);

}

numbers.add(0, first);

return max;

}

}

## Exercise 12.22: printSquares

**Question**

Write a method printSquares that uses recursive backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example, the call of printSquares(200); should produce the following output:

1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2

1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2

1^2 + 2^2 + 5^2 + 7^2 + 11^2

1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2

1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2

2^2 + 4^2 + 6^2 + 12^2

2^2 + 14^2

3^2 + 5^2 + 6^2 + 7^2 + 9^2

6^2 + 8^2 + 10^2

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your method should produce no output. Keep in mind that the sum has to be formed with unique integers. Otherwise you could always find a solution by adding 1^2 together until you got to whatever number you are working with.

As with any backtracking problem, this one amounts to a set of choices, one for each integer whose square might or might not be part of your sum. In many of our backtracking problems we store the choices in some kind of collection. In this problem you can instead generate the choices by doing a for loop over an appropriate range of numbers. Note that the maximum possible integer that can be part of a sum of squares for an integer n is the square root of n.

Like with other backtracking problems, you still need to keep track of which choices you have made at any given moment. In this case, the choices you have made consist of some group of integers whose squares may be part of a sum that will add up to n. Represent these chosen integers as an appropriate collection where you add the integer i to the collection to consider it as part of an answer. If you ever create such a collection whose values squared add up to n, you have found a sum that should be printed.

To help you solve this problem, assume there already exists a method printHelper that accepts any Java collection of integers (such as a list, set, stack, queue, etc.) and prints the collection's elements in order. For example, if a set s stores the elements [1, 4, 8, 11], the call of printHelper(s); would produce the following output:

1^2 + 4^2 + 8^2 + 11^2

**Solution:**

// Prints all ways to express n as a sum of squares of unique integers.

// Precondition: n >= 0

public static void printSquares(int n) {

Set<Integer> chosen = new TreeSet<Integer>();

explore(n, 1, chosen);

}

// Explore all ways to form n as a sum of squares of integers starting

// with the given min and storing the chosen results in the given set.

private static void explore(int n, int min, Set<Integer> chosen) {

if (n == 0) {

printHelper(chosen); // base case: sum has reached n

} else {

// recursive case: try all combinations of every integer

int max = (int) Math.sqrt(n); // valid choices go up to sqrt(n)

for (int i = min; i <= max; i++) {

// try all combinations that include the square of this integer

chosen.add(i);

explore(n - (i \* i), i + 1, chosen);

chosen.remove(i); // backtrack

}

}

}