Lab Report 05

Nathan Bickel

Problem

The problem was, given a list of integers, to recursively print every possible combination of them and its corresponding sum. So, given {1,2}, the program would print (in no particular order):

1 = 1
1 + 2 = 3
2 = 2

Thus, repeats where the order of the components was different but the chosen numbers were the same were not printed. Additionally, the case where no numbers from the list are chosen is ignored (as it wouldn't print anything). The problem also specified that the list should have 5 elements, although it makes sense to write the program in a way that any number of elements could be given (as long as the number isn't unreasonably big).

Solution

Background

The solution uses a bit of math that may not be self-explanatory, so it is elaborated on below:

First, consider the case where there is only one element in the list, such as {1}. The only combinations possible are to include 1 or not include 1. This can be represented in an matrix like this:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

This matrix should be interpreted as a boolean array, with 0 representing false and 1 representing true. So, the first row is the case where the 1 is not chosen, and the second row is the case where it is. Since we're not interested in the case where no numbers are chosen, the first row is ignored and the result we want is for

1 = 1

to be printed.

A case where there are two elements in the list, such as {1,2}, can be represented similarly:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

As before, we ignore the first row. In the second row, we do not choose the first element in the list but choose the second, resulting in choosing 2. In the third row, we choose the first element but do not choose the second, resulting in choosing 1. In the fourth row, we choose both the first and second elements, resulting in choosing both 1 and 2. Thus, we want

1 = 1
2 = 2
1 + 2 = 3

to be printed. This is every possible combination of a list with two elements.

For any n, $n \in \mathbb{R} \mid n > 0$ elements, all combinations can similarly be represented by a matrix. It will have n columns, because each element in a row corresponds to whether or not an element in the list should be chosen in the combination. Because we aren't interested in the case where no elements are chosen, it doesn't make sense to include this case in the matrix (even though it's included in the examples). Thus, the matrix will have $\sum_{k=1}^{n} \binom{n}{k}$ rows.

This is because the matrix needs to have a row for every possible combination, so we add every "n choose k" from k = 1 to k = n:

$$\sum_{k=1}^{n} = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = \frac{n!}{1!(n-1)!} + \frac{n!}{2!(n-2)!} + \cdots + \frac{n!}{n!(n-n)!}$$

Using the recursive definition of binomials, it can be shown by induction that:

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$$

Thus, for our length of interest, 5, n will be 5, and the corresponding matrix will have $5 = 5$ columns and $2^5 - 1 = 31$ rows.

If we then number each row of the array, starting with 1, we can simply let that row of the array be related to the binary representation of its index. As an example, the first three rows from the n = 5 matrix are shown:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

The first row is labeled as index = 1. This can be represented in binary as 00001, which corresponds to the elements in the first row. The second row is index = 2, in binary 00010, and the third row is index = 3, in binary 00011, which follow the same pattern. Similarly, the last row of the matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

is indexed as 31, which is 11111 in binary, again corresponding to the elements in the matrix. Thus, a matrix containing instructions on how to choose combinations from a list can be generated fairly routinely given a length n of the list, and it will be guaranteed to contain every combination.

Implementation

The core to the solution is generating the matrix, so a static double boolean array is declared. Each element corresponds to the same element in the matrix, with false being equivalent to 0 and true being equivalent to 1. In the main method, each boolean

is initially set to false. A static final ***ARR_LENGTH*** was also declared to hold the length of the list, and it was set to 5.

Two helper methods were declared: **printSum** and **generateArray**. **printSum** takes an array as a parameter, prints each element separated by a plus sign, an equal sign, and the sum of the elements. **generateArray** creates an array of the given length, and populates it with random values in a given range (set to 0 to 99, inclusive). The array of numbers to be processed is generated with the **generateArray** method.

The method **recursiveSum** is what prints the combinations. It takes two parameters, an integer called index and an integer array. The index is initially passed as 0, and the array is the array generated by the **generateArray** method. The (index + 1) (since we don't care about the case where no numbers are chosen) is parsed to a String representing its binary representation, and that String is split into an array with each element being a character. It is then mapped to the boolean array, with any extra values on the left being left as false. Then, the boolean array is applied to the list, choosing values in the array for the combination at the same indices that are true in the boolean array. These values are then entered into a new array that is passed to the **printSum** method. Finally, **recursiveSum** is called recursively with index + 1 and the same array. It keeps doing this until it hits the halting condition, which is index >= 2^***ARR_SIZE*** - 1 (as discussed above). Once it hits the halting condition, it returns and the main method is finished.

For example, if **recursiveSum** is given index = 21 and an array of {1,2,3,4,5}:

- index + 1 = 22 will be converted to a String equal to "10110"
- The String will be split to an array equaling {"1","0","1","1","0"}
- The boolean array at row index = 21 will be mapped to {true, false, true, true, false}
- This means we should choose the first, third, and fourth values from the list, so a new integer array is created equalling {1,3,4}
- This array is passed to **printSum**, which prints "1 + 3 + 4 = 8"
- **recursiveSum** is called with index = 21 + 1 = 22 and an array of {1,2,3,4,5}, and the process continues repeating until index = 31

In this way, each combination of a given list is called.

Implementation Problems Encountered

It took a while of thinking to come up with this approach. While it doesn't exactly use backtracking like the assignment suggests, it does (as far as I can tell) meet all the requirements and runs recursively.

It also took a bit of experimenting to figure out how to convert an integer to a binary array, but I eventually figured it out. Other than that, I didn't really encounter any problems—the code is much more straightforward than the concept it implements.

Lab Report Questions

1. Both recursion and iteration have a starting place, an ending place, and a way to get from the starting place to the ending place. In recursion, the starting place is the parameters that are entered in the first call of the method, the ending place is the halting condition that should be written in the method to get it stop when a condition is true, and the way it gets from the start to the end is usually one of the parameters changing from call to call. In iteration, the starting place is the first part of a for statement, the ending place is the second part, and the way it gets from the start to the end is the third part. In both cases, it causes problems if there is no progression to the end—in recursion, this will eventually end in a stack overflow error, and in iteration, this will result in an infinite loop, usually causing a crash eventually.
2. The difference between recursion and iteration is iteration only occurs in one method on the stack, whereas recursion uses the stack more than once (and often many times). Recursion is best used as a divide and conquer approach, where a problem is divided into smaller, easier problems that can be used to solve the bigger one. It is often more difficult to take this approach with iteration.