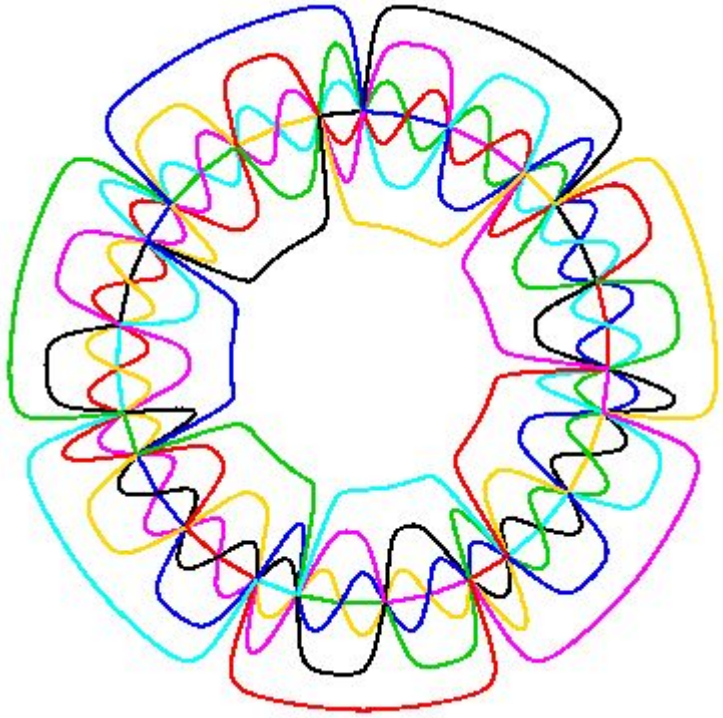
Section 8 - An Algorithm

1A.8.1 The Subset Sum Problem



Besides learning about data structures, this class will explore many ***algorithms***. Some algorithms are used within the data structures and arise as natural support for those structures. Other algorithms are solutions to problems in their own right, and the data structures are used as a means to an end -- as an expedient toward implementing the algorithm. As a taste of this, let's meet an algorithm that will make use of a simple ***vector*** data structure. It stems from something called the ***Subset Sum Problem***.

A version of the ***subset sum problem*** is a pair **(*S*, *t*)**, where $S = \{x_1, x_2, \ldots x_n\}$ is a set of positive integers and ***t*** (the **target**) is a positive integer. The problem asks for a subset of ***S*** whose sum is as large as possible, but not larger than ***t***. The problem has many useful applications. In transportation, we may have a truck that can carry at most ***t*** pounds, and we wish to maximize the load based on a loading dock filled with packages with varying discrete weights. A radio show host or program manager may want to select group of tunes or commercials from a set whose total running time is as close to the duration of the show or commercial break (perhaps three hour FM music show, or a two minute commercial break) without going over the time allotment. The left-over time can easily be filled with talk or station identification.

1A.8.2 The Algorithm - Simple Formulation

- Form all possible subsets (aka "***sub-lists***" of **S**, also called the ***"power set of S"***).
  For example, if **S = {4, 1, 7}**, the **power set of *S*** is {} (the empty set), {4}, {1}, {7}, {4, 1}, {1, 7}, {4, 7}, {4, 1, 7}.
- Find the subset (***sub-list***) whose members add up to the largest number possible $\leq$ ***t***.

This yields a solution (although it may not be unique - there may be many sub-lists that add to the same **maximal number $\leq$ *t***). It is a brute force method of getting the answer. It is a very costly technique in terms of timing because it has what we call ***exponential time complexity*** (we'll learn what this means in two weeks). However, the problem is inherently exponential, so there is not much we can do about that. We *can* still find more efficient algorithms, and the algorithm we give next will make a modest efficiency improvement. It will also explicitly state how we form the power set of ***S***. Before giving it, here is a little terminology that will assist us.

Notation

If ***L*** is a list of positive integers from ***S*** (i.e., a sub-list) and **x** is member of ***S***, then we let ***L*** + ***x*** denote the augmented sub-list derived from ***L*** by adding (i.e., appending) ***x*** to the set ***L***. For example, if ***L*** = **{4, 1, 7}** then ***L*** + **2 = {4, 1, 7, 2}**

sum(*L*) will mean the sum of elements in set ***L***. So, **sum({4, 1, 7}) = 12.**

Col will be some collection of sub-lists. For example, if ***S*** = **{4, 1, 7}**, ***Col*** might be { **{}, {4, 1}, {4, 7}** } (we count the empty sub-list, **{}** as a list in the collection if we want it there).

An exact, but not particularly fast, algorithm to find the sub-list **L'** with the **sum(L')** as large as possible but less than or equal to ***t*** is as follows:

- initialize the collection ***Col*** with one sub-list: the empty sub-list.
- loop over all elements ***x*** in ***S***
    - loop over all sub-lists, ***L***, that are already members of ***Col***
        - if **sum(*L*) + x** $\leq$ ***t***, add the ***sublist L* + x** to ***Col***
        - if **sum(*L*) + x == *t***, break from both loops
- of the sub-lists that end up in ***Col***, find the sub-list ***L'*** with the largest **sum()**

The set ***L'*** in the last step is the sub-list which has the largest sum, without going over the **target**, ***t***.

There are two comments I will add about the algorithm:

1. **Clarification**. The inner loop over ***Col*** does not have a growing upper limit if you end up adding sub-lists inside the loop. If there are 143 sub-lists in L going into the loop, you only loop over those 143 sub-lists. Any new sub-lists added to ***Col*** are not part of that loop pass but are only considered the next time this loop is entered from above.

2. **Efficiency**. You can and should test in your inner loop to see if you have added a new sub-list whose sum() == t, because if you have, you are done and should break out of this outer loop. Normally there will be zillions of sub-lists and we can usually reach the target long before we have exhausted them all.

We will see how one can use ***vectors*** (coming up next time) to assist us in finding an implementation of the ***subset sum problem***.

Exercise (DO NOT SKIP OR YOU WILL NOT BE ABLE TO DO THE HOMEWORK): *Using a pencil and paper, pick a random set of 10 integers ranging from 3 to 20. Use the algorithm above to solve the subset sum problem for your set and the target 37.*