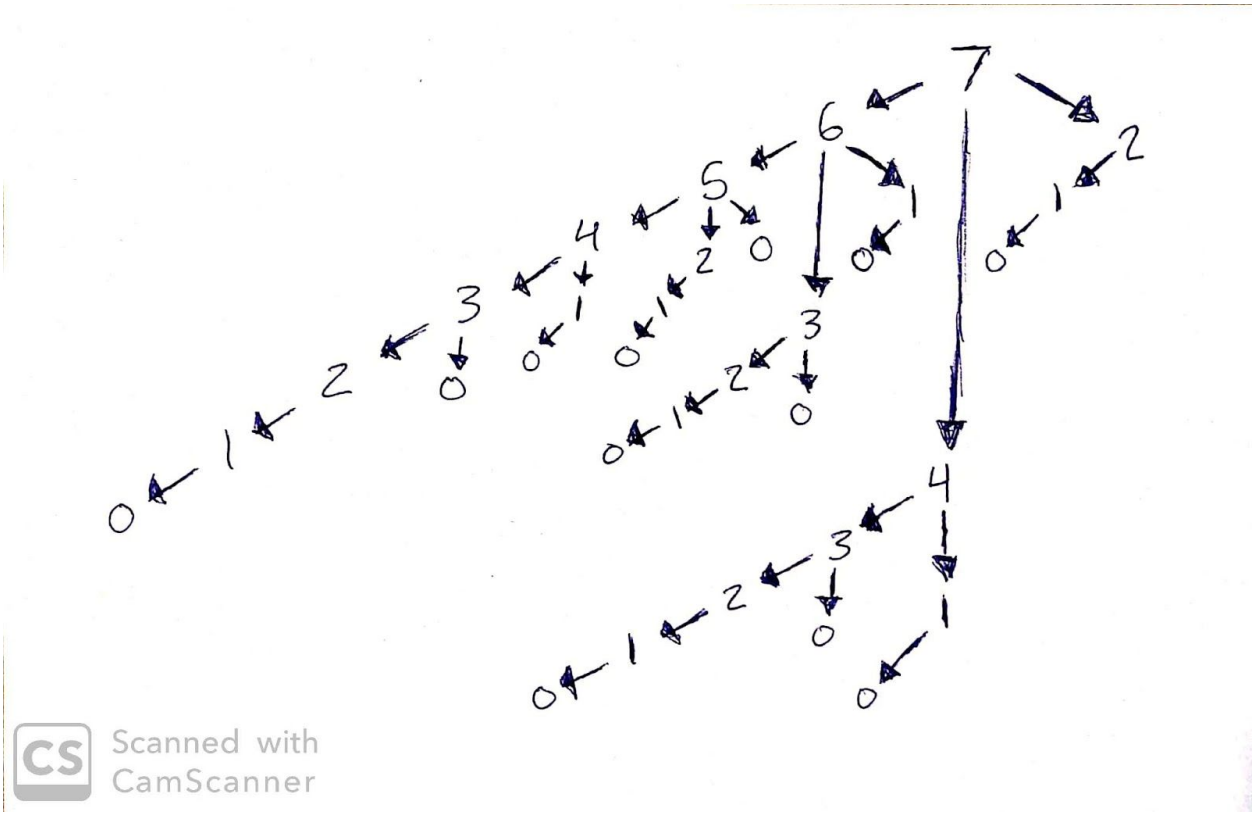
Project 2

1. How you can break down a problem instance of the "order matters' problem for a given target t and array data into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the subproblems.

To solve this problem you have to make a map. This is done by subtracting t by each value in data. If that solution is greater than or equal to 0, you make it a child. Then you do the same with your new children. This continues until the children can no longer be subtracted. By the end of this process all the zeros found will be all the possible solutions.

Graph for data = [1,3,5] and t = 7 :



This graph shows the repeats we have in our solution. This now lets us use a 1d map in order to memoize this algorithm

2. What are the base cases of the "order matters" recurrence?

The Base cases for this dynamic algorithm are:

1. If when subtracting you obtain a 0

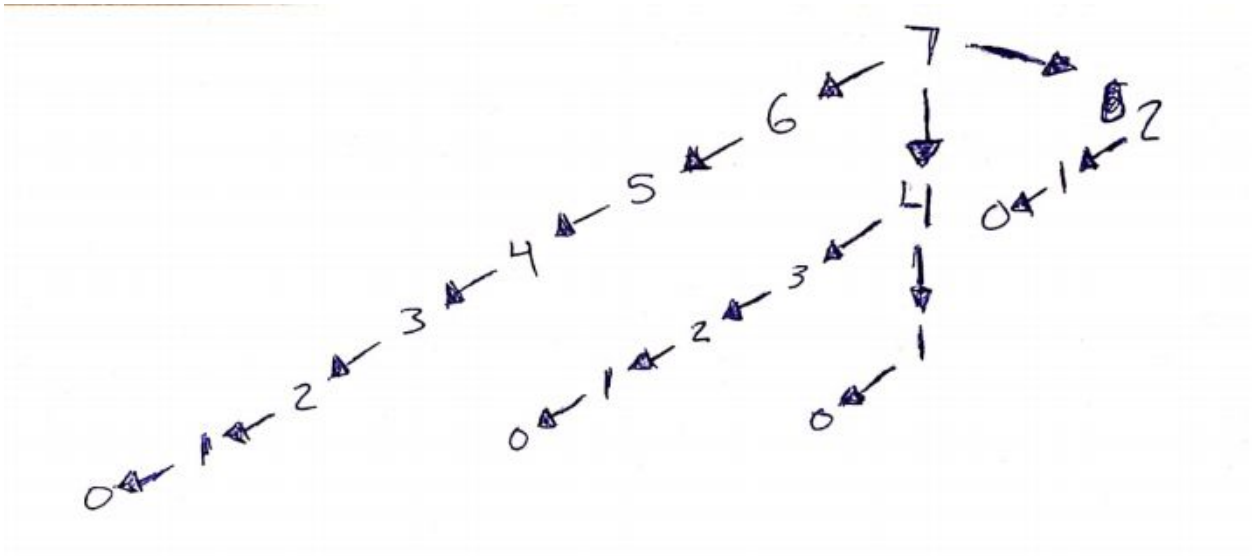
2. If the current value are at is already in the map

There is also a check to make sure the value - current data member is >= 0, as the value passed to the recursive calls can't be negative.

3. How you can break down a problem instance of the "order doesn't matter" problem for a given target t and array data into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the subproblems.

To solve this problem it is also helpful to make a graph. Instead of calling every time your new value will be >= 0, you also check to make sure that the numbers you are subtracting are in order. To achieve this we check to only make a recursive call with values that are less than values already called.

Graph for data = [1,3,5] and t = 7 :



This graph shows the solutions:

1. 1+1+1+1+1+1+1
2. 3+1+1+1+1
3. 3+3+1
4. 5+1+1

All solutions are in order from greatest to least. This also shows that we need a 2d map. This is because two variables are now changing, the current value and the data member we used to get to this value. This is shown well by the value four. When getting to four by subtracting ones we get one solution. However when getting to four by first subtracting three we get two solutions.

4. What are the base cases of this "order doesn't matter" recurrence?

The base cases for this dynamic algorithm are:

1. If map at current value and number you are subtracting is already in map
2. If when subtracting you obtain a 0

There is a check to make sure the value - current data member is >= 0 and a check to make sure you are only making ordered solutions.

5. What data structure would you use to recognize repeated problems for each problem? You should describe both the abstract data structures, as well as their implementations.

For "order does matter":
To recognize the repetitions we drew out what the map would look like for the problem. Starting from t, subtracting each value in data until we reach a 0 at the final child. This showed that we were calculating same solutions multiple times. The data structure we used to fix this problem was a direct map. To use the map we set the map[t] += recursion and return map[number] if the value at the place doesn't equal the sentinel value. This makes it so if we have already calculated that value we can pull it from the map instead of calculating it again.

For "order doesn't matter":
We also drew a tree out for this solution. The difference was that we knew we wanted only sorted data set to get rid of duplicates. To achieve this we need a two dimensional map. This gives us the opportunity to store different types of the same value. For example, with a data set of [1,3,5] and a t of 7 there is two ways to get to value 3. They are 1+1+1 and 3. We can store these in seperate places, array[3][0] for when adding ones, and array[3][1] for adding threes.

In both cases we implemented the maps with an dynamically allocated array, using the C++ Vector class.

6. Give pseudocode for a memoized dynamic programming algorithm to find the number of sums when order matters

Helper function(t)

1. Map = Array(t+1)
2. Initialize map to 0
3. Order Does Matter(t)

Algorithm: OrderDoesMatter(t)

1. If map[t] != 0

2. Return map[t]
3. If t == 0
4. Set map[t] = 1
5. For i = 0 to data.length -1 do
6. If t - data[i] >= 0
7. map[t] += OrderDoesMatter(t - data[i])
8. Return map[t]

7. What is the worst-case time complexity of your memoized algorithm for the order matters problem?

O(nt). It takes O(t) to initialize the map. the loop runs for O(n). The cost of memoization is T= #cells*costpercell. This yields T = t * n or O(nt), which is larger than initialization.

8. Give pseudocode for a memoized dynamic programming algorithm to find the number of sums when order doesn't matter

Helper function(t)

1. map = array(t+1)(data.length - 1)
2. Initialize map to 0
3. OrderDoesntMatter(t, data.length - 1)

Algorithm: OrderDoesntMatter(t, subtractor)

1. If map[t][subtractor] != 0
2. Return map[t][subtractor]
3. If t == 0
4. Set map[t][subtractor] = 1
5. For i = 0 to data.length - 1 do
6. If t - data[i] >= 0 && data[subtractor] >= data[i]
7. map[t][subtractor] += OrderDoesntMatter(t - data[i], i)
8. Return map[t][subtractor]

9. Give pseudocode for an iterative algorithm to find the number of sums when order doesn't matter. This algorithm does not need to have a reduced space complexity.

Helper function(t, data)

1. map = array(t+1)(data.length - 1)
2. Initialize row 0 of map to 1's
3. Initialize rest of map to 0's
4. OrderDoesntMatter(t, data)

Algorithm: OrderDoesntMatter(t, data)

```
1.  for i=1 to t do
2.      for j=data.length -1 to 0 do
3.          K = data.length -1
4.          While (j <= k)
5.              if(i - data[k] < 0)
6.                  Map[i][j] += 0;
7.              Else
8.                  Map[i][j] += map[i-data[k]][k]
9.              K--
10.         End
11.     End
12. End
```