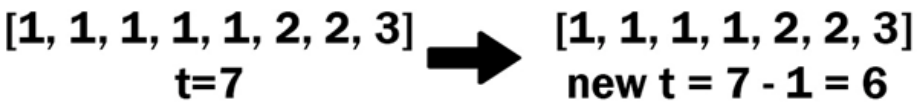
**Project 2 Report: Subset Sum Problem**

Group members:
Matias Cinera
Iban Cruz
Zachary Zweibach

1. How you can break down a large problem instance into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the subproblems and why this breakdown makes sense.

The goal of the subset sum problem is to find all the possible subsets which sum equals to a target. The way to break this problem into smaller instances is to remove elements from the set. Whenever you remove an element from the set you are essentially creating a subset, the missing element from the subset will represent part of the sum thus a new target should be calculated. The new target will equal the old target minus the removed element.

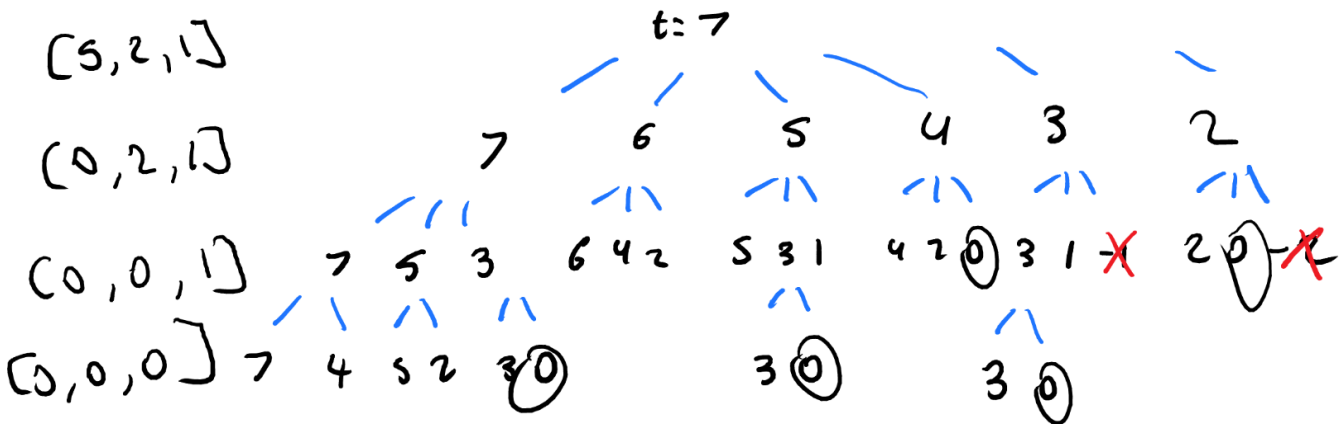
Ex: Given a set of elements with 5 ones, 2 twos, 1 three and your target sum is 7. If you remove a 1 from the original set, you are 1 step closer to the sum, thus the new target for this subset will be 6 instead of 7.




Now you can repeat this process with any subset, remove an element and update the target, thus breaking the problem into smaller subsets and target values (smaller instances). Eventually, you will find a subset which's target value is zero, less than zero or an empty subset, all of these mean that you should not remove any more elements from that subset. If a subset target equals 0, this means you found a solution ,therefore, you can count that subset as one solution to the original set (technically the solution will be the complement of the subset). If a subset target is less than 0, it means that you went over the original target. The answer to the original problem will be counting the number of subsets which target equals 0.

Note: To split the original set into multiple subsets you will need a proper parsing algorithm to uncover all possible paths. This is an example of how of a parsing tree represented by the target:

```
Given the set = {1, 1, 1, 1, 1, 2, 2, 3} or {5, 2, 3} and target = 7
{5, 2, 3} is the index representation of the same set
```




Therefore this set will have 5 different solutions.

2. What recurrence can you use to model this problem using dynamic programming?

This recurrence has a different number of calls in each recursive instance. The number of recursive calls depends on the frequency of a given number in the set. For example, if you have 5 ones in the set, you will have 6 different recursive calls (one extra call to consider when the sum does not include any 1s). No matter the number 1s you add to the recursive solution, the next recursive call will be the following index of the set. However, the algorithm can only recur up to t/index times. Since if the target is 5 it can only recurse up to 5 times for index = 1, since it will then equal the target and can no longer add values. For index = 2 it can only recur up to 2 times, since adding more than two 2's would result in the sum exceeding the target. Which would stop the recursion.

So, the number of recursive calls per iteration will be up to: t / index, index goes from 0 to n.
Also, the outside code is O(n).
Therefore, the recurrence is:

$$\sum_{i=1}^{n} \frac{t}{i} + O(n)$$

3. What are the base cases of this recurrence?

The recurrence has four base cases:

1. If the memoized data structure already contains a map representing the remaining numbers in data and if this map has a value for the current target sum. This means the number of sums in this subset = target has already been calculated. It should return the value stored.
2. If the current target value is equal to 0. This subset's sum equals the target sum, it should return 1 to indicate this.
3. If the target value is less than 0. This subset's sum is greater than the target sum, there is no reason to add more numbers. 0 should be returned.
4. If the size of the data map is empty. There are no more numbers to be added to the subset. 0 should be returned.

T(memo check) = Θ(1)
T(sum of set = target) = Θ(1)
T(sum of set > target) = Θ(1)
T(set is empty) = Θ(1)

4. What data structure would you use to store the partial solutions to this problem? Justify your answer.

The ideal data structure for this algorithm will be a map. The key and value of this map will be another two maps, both key's and value's maps will have integers to store their respective keys and values.

```
map<map<int, int>, map<int, int>> memo;
```

Let's say *data* represents a set of numbers where the mapped value is the quantity of the key value present in the set.

For example, *memo*[3, 1, 0] is a set with 3 ones and 1 two. We would call *sum* on the second map (the value of the *memo*). The second map tracks how many different subsets add to a *sum*. *Sum* is the key and the mapped value is the number of subsets. Continuing the example memo [2, 1, 1][3] would equal to 2. Since there are two subsets that add up to 3. [0, 0, 1] and [2, 1, 0].

5. Give pseudocode for a memoized dynamic programming algorithm that uses memoization to compute the number of subsets that sum to t.

Note: The code has a *lot* of comments explaining what it does and how it works. I feel that this should allow the pseudocode here to be more english like rather than code like.
input:
target - sum desired for the function
data - map of ints from 1 to n. The mapped value is the quantity of the key value in the data set. For example, if data[1] == 5, there are five 1's in the data set.
n - largest key in data (Note, this actually isn't used or needed in our program. At all.)

note:
memo - a map of maps for memoization purposes. It is explained both in the commented code and in #4 of the report. Memo stores, for a map Data and for a sum Target, how many subsets of Data add up to Target. For example, if data = [2, 1, 1] and target = 3, memo[data][target] = 2. Since there are two subsets of data that add up to 3, [2, 1, 0] and [0, 0, 1].

Output: the number of subsets from data which sum equals to the target.

```
      The number of subsets from data where their sum equals target.
1.  Algorithm: WrapperCS
2.  Initialize memo as previously described
3.  return CountSubsets(target, data)

1.  Algorithm: CountSubsets
2.    (The next three if statements are base cases)
3.  if [data][target] exists in memo
4.  return it
5.  end
6.  if target is 0, data sums to target perfectly.
7.  set memo[data][target] = 1
8.          return 1
9.  end
10. if target is negative or data is empty. Data's sum will never equal t.
11.          set memo[data][target] = 0
12.          return 0
13. end

14. leftmost = data.begin (which will be the smallest value)
15. small = data.first (the actual value of the smallest value)
16. sAmt = data.second (the quantity of the smallest value present in data)

17. d = copy of data, removing the smallest value. (Note that d is a map so this won't change
indexes or anything)

/*The number of subsets in data whose sums = target is stored in memo[data][target].
We now need to calculate how many that is, so we will recurse sAmt times and add the value returned to
memo[data][target]
Each recursion will have [0, 1, 2, ..., sAmt] small subtracted from target.
Because if we have 1 small in the subset, we only need to get target - small from the rest of the subset.*/

18. memo[data][target] += recurse on target and d. This represents zero small
being added.

19. sum = 0
20. for i = 1 to sAmt
21. sum += small
22. memo[data][target] += recurse on target - sum and d. This represents anywhere from
                                                                    0 to sAmt small being added.
23. end
24. return memo[data][target]
```

6. What is the time complexity of your memoized algorithm? Show your work.

```
      1.  Algorithm: WrapperCS
O(1)  2.  Initialize memo as previously described
???   3.  return CountSubsets(target, data)

      1.  Algorithm: CountSubsets
      2.
O(lgn)3.  if [data][target] exists in memo (.find)
O(1)  4.  return it
      5.  end
O(1)  6.  if target is 0
O(1)  7.  set memo[data][target] = 1
O(1)  8.          return 1
      9.  end
O(1)  10. if target is negative or data is empty
O(1)  11.           set memo[data][target] = 0
O(1)  12.           return 0
      13. end
O(1)  14. leftmost = data.begin
O(1)  15. small = data.first
O(1)  16. sAmt = data.second
O(n)  17. d = copy of data, removing the smallest value
Rec(1)18. memo[data][target] += recurse on target and d.
O(1)  19. sum = 0
Loop1 20. for i = 1 to sAmt
O(1)  21.   sum += small
Rec   22.   memo[data][target] += recurse on target - sum and d
      23. end
O(1)  24. return memo[data][target]
```

Summary:

Wrapper: The wrapper initializes our memoization map, which is O(1), then calls the recursive function. The recursive function will obviously overshadow O(1). So the complexity is: BLANK

Outside Loop1: Outside loop 1, the complexity is O(n), from copying data. We also recurse one time outside the loop.

Inside Loop1: Inside loop 1, the nonrecursive code is O(1). Which again will be overshadowed by our recursive call.

Loop 1 will run sAmt - 1 times, so the function will recurse that many times. However it also recurses once outside the loop. So it will recurse sAmt times per call where sAmt = the quantity of the smallest number. So, the number of recursive calls per iteration will be: data[index] per iteration
Since you have n elements, you will have n calls of data[index]. However, the calls will stop when the value reaches t. Since the smallest possible value is 1, the first call will have up to t calls. Since the second smallest possible value is 2, the second call will have up to t/2 calls. Since the third smallest possible value is 3, the second call will have up to t/3 calls. So, there would be t + t/2 + t/3 +...+t/n calls. This summation will, at the max, round to 2t. So we have 2t calls and n complexity per call. Which leads to an end complexity of 2t * n, or t * n. So, our algorithm has a complexity of O(t * n).

7. Give pseudocode for an iterative dynamic programming algorithm for this problem.

Even though the data structure used for memoization is a really complicated one (a map with maps as keys and values), our algorithm recursion only relies on updating the index for the next recursive call. This was done by creating a temporary map and deleting the smallest key, however, this can also be achieved with a loop that runs through every index in data. If you want to calculate memo[data][target], this call will rely on memo[old _min+data][target] ), which is the previous index. This means the order of iteration will be from Left to Right. This means all the values memo[data][target] depends on will already be calculated.

Note: based on the memoized algorithm

```
Algorithm: CountSubsetsIterative
1. Initialize memo
2. count = 0;
3. for i = 0 to n
4.    if [data][target] exists in memo
5.          count = count + [data][target];
6.    end
7.    if target is 0
8.          set memo[data][target] = 1
9.          count = count + [data][target]
10.   end
11.   if target is negative or data is empty.
12.         set memo[data][target] = 0
13.   end

14.   leftmost = data[i]
15.   small = map[i] key
16.   sAmt = data[i] value

17.   memo[data][target] += memo[data][target]
      ^This represents zero small being added.
18.   sum = 0
19.    old_target = target
20.   for i = 1 to sAmt
21.         target = target - sum
22.         sum += small
23.         memo[data][old_target] += memo[data][target]
24. end
25. return memo[data][target]
```

8. Can the space complexity of the iterative algorithm be improved relative to the memoized algorithm? Justify your answer.

We can improve the space complexity of the iterative algorithm by using a map with a fixed size equal to the frequency of the key in data with the most values. This improves the space complexity of the iterative algorithm since instead of storing nt possible elements we only need to store the highest frequency of n. After each iteration we can just override the values of the map with fixed size and the algorithm would still work.