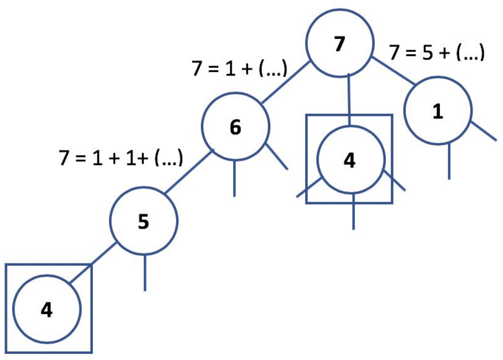
Project 2: Dynamic Programming

Report

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.

For the "Order Matters" algorithm, our team developed a naive recursive algorithm to describe how to reach the solution. The first observation that was made, is that to get the target value from one of the elements in the input array, you have to add that element in the array plus another sum. That new sum became the new target value in the recursive call. To get that new target value, the current element in the array was subtracted from the original target value and passed as the new target value in the recursive call. Therefore, the solving the recursive sums for each element in the array broke down the initial problem into subproblems. We determined that the most systematic solution was to subtract all the values in the input array from a target value using a for loop, and continue to do so recursively until zero or less than zero was the target value passed. The summation of the number of times a zero was found at each step was stored and returned from the algorithm at each call. The following figure demonstrates how our group broke down the problem instance:



From the recursion tree of the example that was drawn out, we noticed that there were overlapping subproblems. Since, dynamic programming is defined as recursion augmented with a map to save sub results, we developed a more optimized memoized algorithm.

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

The base cases for order matters in the naive algorithm are if the target value equals zero then return one or if the target value is less than zero then return zero. In the memoized algorithm, an additional base case is included that checks to see if the target value is already solved in the data structure, if so return the value stored at that position in the array.

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 break down the problem instance of the “order doesn’t matter” problem, our team looked how the elements in the array could be added to get the target value. Similarly, to the “order matters” problem we developed a naive recursive algorithm. For this problem to avoid duplicate combinations of elements in different orders to get a target value, we passed the current value of the array. This was the major deviation from our previous “order matters” algorithm and forced the sums to be ordered. Instead of looking for a sum that included all the values in the array, only the values that were next in the array were looked to to see if that order of sums would produce the target value.

4. What are the base cases of this “order doesn’t matter” recurrence?

For the naive recursive algorithm there were two base cases, $t = 0$ and $t < 0$. In the memoized version of the “order doesn't matter an extra base case was added that included the base case where if the value in the two dimensional array was greater than zero then that value was returned.

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 the “order matters” problem presented in this project, our team decided on using a one dimensional array of size N + 1, where N is equal to the target value. The problem, the sentinel value of all the elements in the array are initialized to zero. For the “order doesn't matter” a two dimensional array was used of size n by t+1, where n is equal to the size of the input array and t is equal to the initial target value passed. The sentinel value of the first column was initized to one and all the other values were initialized to 1.

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

Recursive algorithm:

```
    Input: data array, data size N, target value t
    Output: number of ways to sum elements in data to get target value
    Algorithm: MemoOrderMatters
    if sums[t] > 0
        return sum[t];
    end
    if  t == 0
        return 1;
    end
    if  t < 0
        return 0;
    end
    for int i = 0 to N-1
        if (t-data[i] >= 0)
            sum[t] += MemoOderMatters(data, N, t-data[i]);
        end
    end
    return sum[t];
```

Wrapper function:

```
    Input: target t
    Output: sum[t]
    Algorithm: DPOrderMatters
    sum = Array(N+1);
    Initialize sum to 0;
    return sum[t];
```

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

For order matters worst-case time complexity, there are two questions that must be answered. One is how many cells are being filled in during non-memoized calls. In our teams algorithm, we used a one dimensional array of size t, which corresponds to the initial target value. Therefore, there are Θ(t) cells being filled in during non memoized calls. The next question to determine the time complexity is how long it takes to fill in one cell. In the MemoOrderMatters algorithm above, the cost to fill in one cell is Θ(1) * Θ(n) = Θ(n), because to determine a sum target value, all the elements in the array are considered . Finally, to calculate the total time our team used the equation:

Total time = # cells * cost per cell + initialization cost

$= \Theta(t) * \Theta(n) + \Theta(t)$

$= \Theta(nt)$

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

Recursive algorithm:

```
    Input: target t, array data, data size N, current element curr
    Algorithm: OrderDoesntMatter
    if (sum[x][t] > 0)
        return sum[x][t];
    end
    if t == 0
        return 1;
    end
    if t < 0
        return 0;
    end
    for  i = curr to N-1
        if (t-data[i] >= 0)
            sum[x][t] += OrderDoesntMatter( t-data[i],  N, i);
        end
    end
    return sum[x][t];
```

Wrapper:

```
    Input: target t, data size n, current c
    Algorithm: DPOrderDoesntMatter
    sum = Array(n, t+1);
    Initialize column sum[0] to 1 and all other elements to 0;
    return sum[c][t];
```

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.

```
    Input: target t, array data, data size n
    Algorithm: IterativeOrderDoesntMatter
    found = Array(t +1);
    found [0] = 1;
    temp = 0;
    for i = 0 to n-1
        temp = data[i];
        for j = 1 to t+1
            if (j >= temp)
                found[j] += found[j-temp];
            end
        end
    end
    return found[t];
```