Problem 1

a)

Recursive algorithm:

A simple recursive solution would be to consider all subsets of items and find the total weight and value of all subsets, and then just pick the maximum valued subset that is less than or equal to the size of the knapsack. We will have two cases for each item in the subset:

Case 1: An item is included in the optimal set

Case 2: The item is not included in the optimal set

So to find the max value, we take the max of the following:

1) Max value from n-1 items and weight
2) Value of nth item plus max value from n-1 items and W minus the weight of the nth item

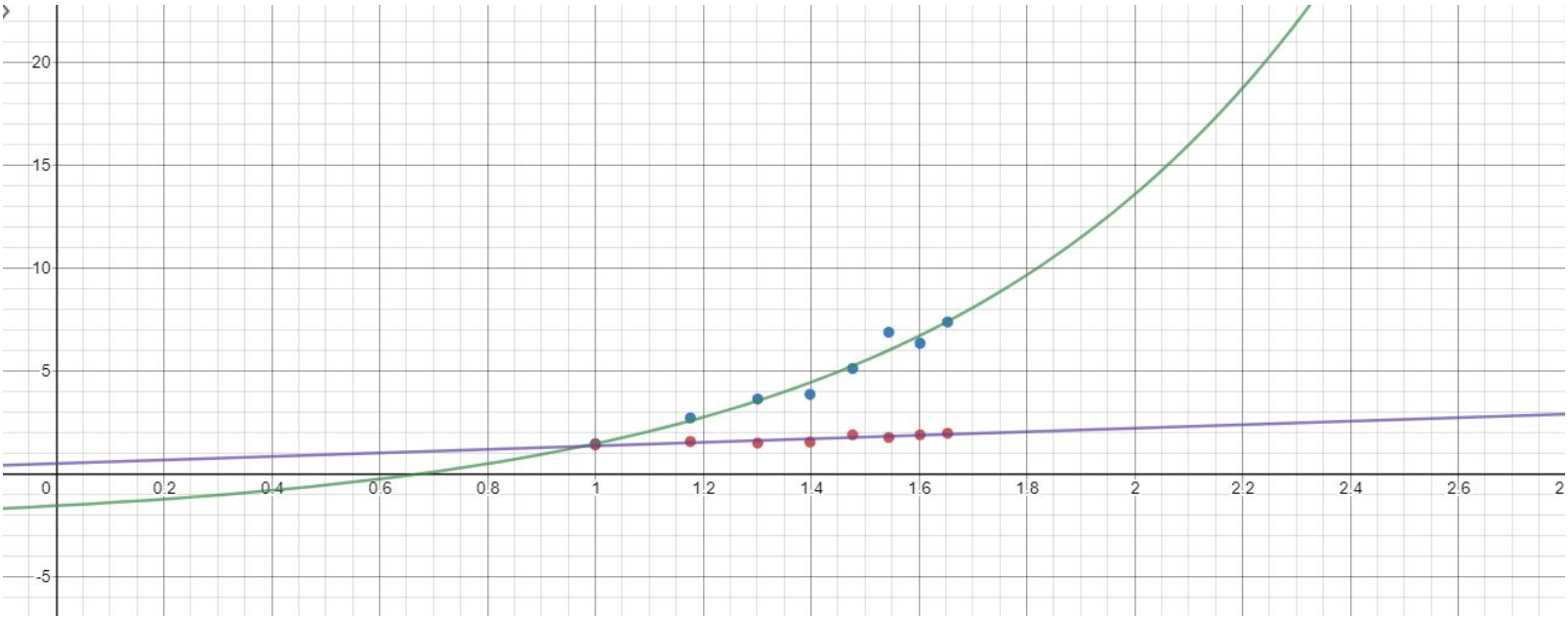
```
function rec_knapSack(W, wt, val, n)
    if n == 0 or W == 0
        return 0 //Base case

    if wt[n-1] > W //If weight of nth item > W, don't include in optimal solution
        return rec_knapSack(W, wt, val, n-1)
    else //return max of the two cases: nth item included and not included
        return max(val[n-1] + rec_knapSac(W-wt[n-1], wt, val, n-1),
                   rec_knapSack(W, wt, val, n-1))
```

DP algorithm:

We can use the tabular method to solve this problem. A w by n table named V can be created such that each cell is calculated by V[n, w] = max{V[n-1, w], V[n-1, w-w[n]]+value[n]}. Thus the bottom right cell will contain the maximum value that can be made in the knapsack.

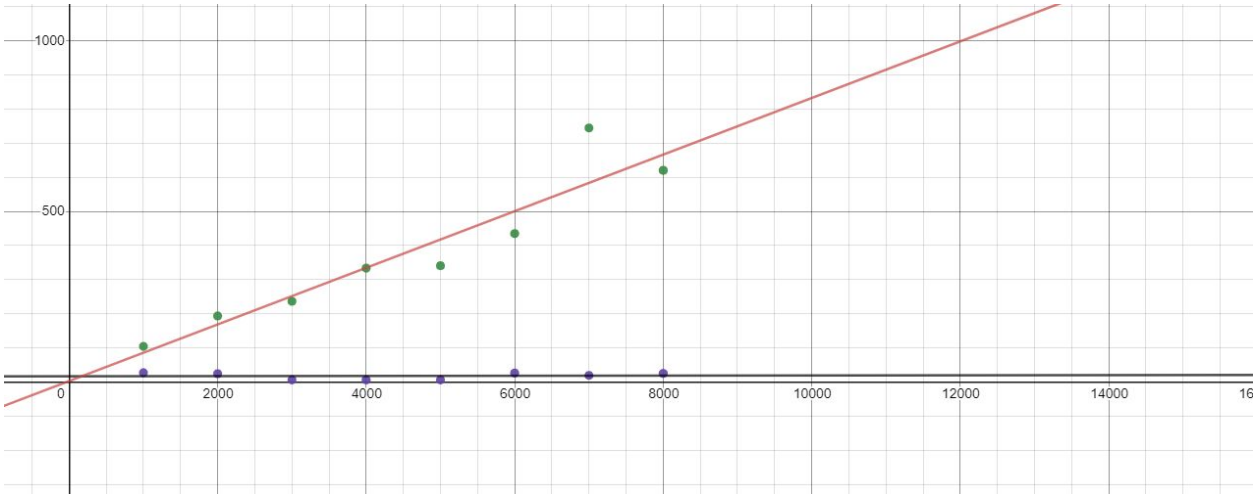
```
function DP_knapSack(W, wt, val, n)
    K[n+1][W+1]
    //Build table K bottom up
    for i from 0 to n+1
        for w from 0 to W+1
            if i==0 or w==0
                K[i][w] = 0
            else if wt[i-1] <= w
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1],
                              (K[i-1][w])
            else
                K[i][w] = K[i-1][w]
    return K[n][W]
```

c) W is constant and N changes (W = 100) (LogLog Graph) (Green = Recursion, Purple = DP)



$$y_1 \sim a_2^{x_1} + b$$
$$y_2 \sim ax_2 \cdot 100 + b$$

N is constant and W changes (N = 10) (Purple = Recursion, Red = DP)



$$y_1 \sim ax_1 + b$$
$$y_2 \sim ax_2 + b$$

d)

In my program, I had two functions. A function that solved the knapsack problem using recursion, and another that solves it using dynamic programming. To collect my data, I ran each of these functions 16 times. The first 8 was with holding the capacity W at a constant 100 varying n from 10 to 45, and the next 8 held n constant at 10 while varying W from 1000 to 9000. I collected the times for each of these runs and plotted them using Desmos. As expected, when holding W constant, the recursive solution had a curve that fitted the O(2^n) runtime, while the DP solution fitted to O(n*W). This changed however when keeping n constant. Since n was constant, the recursive solution would be O(2^C), which would be O(1), and the DP solution would be O(CW), which is O(W). So as W increases, the time also increases linearly.

Problem 2:

a)

This problem can be solved the same way as the knapsack problem, expect rather than just returning the max values that can be carried, we also need to store the items that were used to calculate the sum

```
function DP_knapSack(W, wt, val, n, items)
    K[n+1][W+1]
   //Build table K bottom up
    for i from 0 to n+1
    for w from 0 to W+1
    if i==0 or w==0
        K[i][w] = 0
    else if wt[i-1] <= w
        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1],
                    (K[i-1][w])
    else
        K[i][w] = K[i-1][w]

    //Backtrace to find items that make up knapsack
    max = K[n][W]
    weight = W

    for i = n, i > 0 && max > 0, i--
        if max is not equal to K[i-1][weight]
            push_back i into items
            max -= val[i-1]
            weight -= wt[i-1]
    return K[n][W]
```

b)Since this function runs at most N*M*2 times for each family member, the complexity of this algorithm for all family members is $\Theta(N * M * F)$ Note that this assumes M_small is less than M_max, which is always true.