Problem1:
a)

For Recursive Approach: In order to find the maximum value of the knapsack, we need to consider all subsets of items. For each item, it can either be selected (when weight of item > capacity of knapsack) or not be selected (when weight of item < capacity of knapsack) in the knapsack. First we need to find the size of the items and define the base cases for the algorithm. The base cases are when n=0 and W<0(capacity of knapsack), the algorithm should just return 0 since there are no items to be found. When the weight of item is greater than the capacity of the knapsack, then we call the function itself with parameters of size n-1 and remaining capacity. When weight of item is less than capacity, then we recursively call the function itself with simpler input and return the maximum value. The running time of this algorithm is O(2^n).

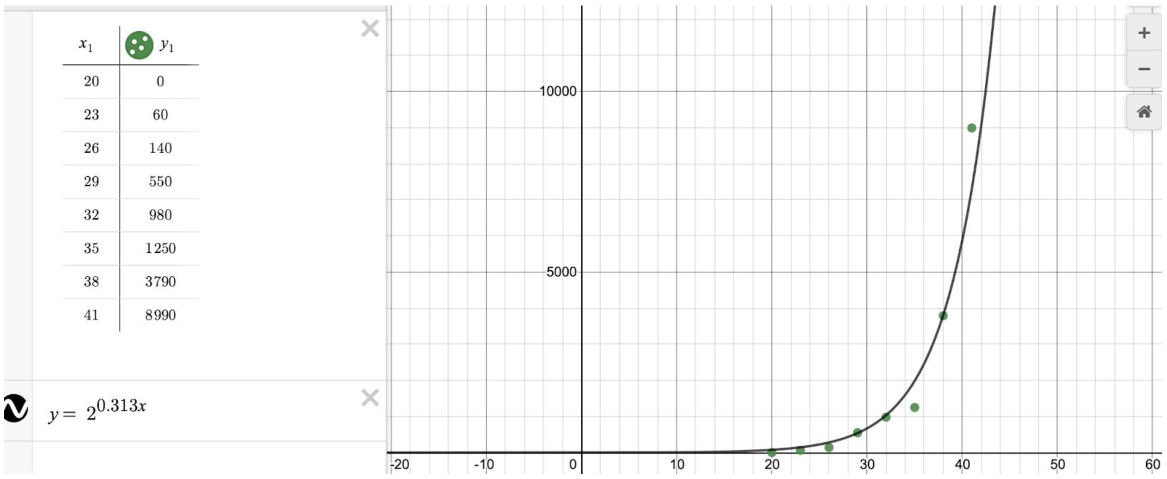
Pseudocode:
```
Knapsack(W, n)
{
        // Base Case
        if (n = 0 or W = 0)
            return 0;

        if (wn > W)
            return Knapsack(W, n-1);
        else
            return max(vn + Knapsack(W-wn, n-1),  Knapsack(W, n-1) );
}
```

For DP Algorithm: another way to solve this problem is the DP algorithm. We can trade in memory for a better time complexity. First we need to create a 2D-array of n+1 rows and W+1 columns and initialize the first row and first column to 0. Array[i][j] denotes the maximum value of j weight considering values from item 1 to item i. The cases are similar to the above algorithm. We will need a 2 nested for loops to loop through each element in the array and calculate the maximum value each element can have in the knapsack. The time complexity for this algorithm is O(nW).

Pseudocode:
```
Knapsack(w,v,n,W)
```

```
{
    Create 2D array V[n+1][W+1]
    for w = 0 to W
        V[0,w] = 0    // 0 item's
    for i = 1 to n
        V[i,0] = 0       // 0 weight
        for w = 1 to W
                if wi <= w // item i can be part of the solution
                        if vi + V[i-1,w-wi] > V[i-1,w]
                                V[i,w] = vi + V[i-1,w- wi]
                        else
                                V[i,w] = V[i-1,w]
                else V[i,w] = V[i-1,w]  // wi > w item i is too big
    Return V[n][W]
}
```

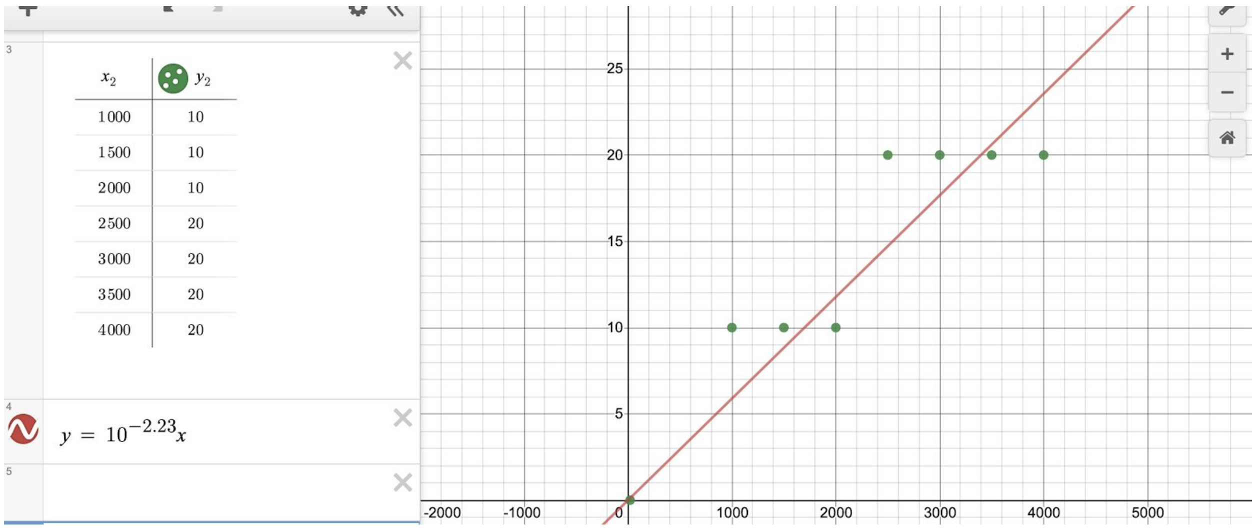
c)



x_1	y_1
20	0
23	60
26	140
29	550
32	980
35	1250
38	3790
41	8990

$y = 2^{0.313x}$

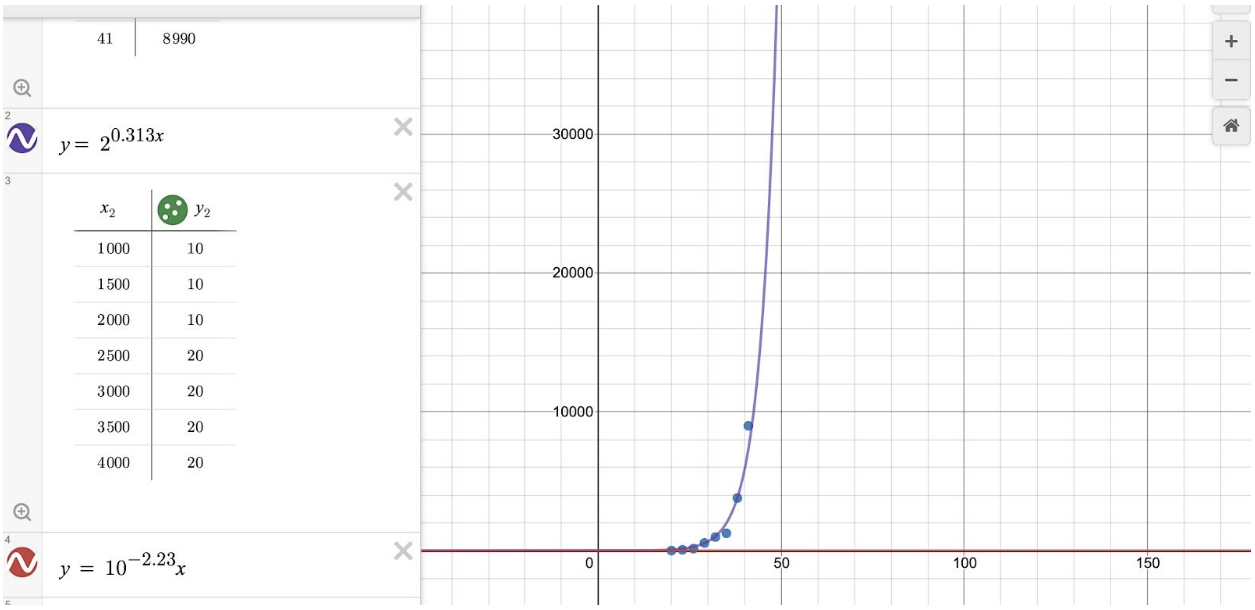
Figure 1: Graph of Running Time for Recursive Algorithm



x_2	y_2
1000	10
1500	10
2000	10
2500	20
3000	20
3500	20
4000	20

$y = 10^{-2.23}x$

Figure 2: Graph of Running Time for DP algorithm



41	8990

$y = 2^{0.313x}$

x_2	y_2
1000	10
1500	10
2000	10
2500	20
3000	20
3500	20
4000	20

$y = 10^{-2.23}x$

Figure 3: Combination Graph of DP(red) and Recursive(purple) algorithms

d)
I created a for loop to run the algorithms with increasing size n starting from 20 to 41 and weight capacity W of 100; each time n was incremented by 3. The running time of the recursive algorithm increased as size n increased, meanwhile the running time of DP algorithm remained 0 since n is significantly small for DP algorithm. Then I created another for loop to run the DP algorithm with larger size n starting from 1000 to 4000 and weight capacity W of 500. I collected data for the algorithms differently and graph them. The experimental running time for recursive algorithm is the same as expected in part a which is O(2^n). The DP algorithm seems to have a linear curve to me. I think this

is because I kept W the same for each run which might make the running time for DP algorithm linear since constant can be dropped in big O analysis.

Problem 2:

a)

This problem is basically similar to the 0/1 knapsack problem. I'd run a for loop T(number of test cases) times. Each time I'd read data from the input file and store it in memory and then do the same thing mentioned in problem 1.

Pseudocode:

```
shoppingSpree()
{
    Read T from input file
    for k = 0 to T-1
        Read N from file     // number of items
        for p = 0 to N-1
            Store price in v
            Store weight in w
        Read F        // number of people in family
        for u = 0 to F-1
            Read M  // max weight that each can carry
            Create a 2D array V[N+1][M+1]
            for w = 0 to M
                V[0,w] = 0   // 0 item's
            for i = 1 to N
                V[i,0] = 0      // 0 weight
                for w = 1 to M
                     if wi <= w // item i can be part of the solution
                           if vi + V[i-1,w-wi] > V[i-1,w]
                                V[i,w] = vi + V[i-1,w- wi]
                           else
                                V[i,w] = V[i-1,w]
                     else V[i,w] = V[i-1,w]  // wi > w item i is too big
    Return V[n][W]
}
```

b)
The running time of my algorithm for one test case is O(NM) where N is the number of items and M is the maximum weight each person can carry.