1) Canoe Problem

For this problem my approach was to first identify the smallest subproblem to be solved, which is to determine, for each distance traveled, the combination of pick up an drop off points that cost the least.

In order to solve this, I defined a new Array C[i], where C[i] is the optimal solution to get to trading post i.

$Base\ Case, travelpost = 1$:

$C[1] = R[1,1] = 0$

$travelpost = 2$:

$C[2] = R[1,2] + C[1]$

$travelpost = 3$:

$$C[3] = min\begin{Bmatrix} R[1,3] \\ R[2,3] + C[2] \end{Bmatrix}$$

$travelpost = 4$:

$$C[4] = min\begin{Bmatrix} R[1,4] \\ R[2,4] + C[2] \\ R[3,4] + C[3] \end{Bmatrix} R$$

$travelpost = n$:

$$C[n] = min\begin{Bmatrix} R[1,n] \\ R[2,n] + C[2] \\ R[3,n] + C[3] \\ R[4,n] + C[4] \\ \vdots \\ R[n-1,n] + C[n-1] \end{Bmatrix}$$

a. Pseudo Code, Bottom-Up Dynamic Programming:

from above we can see that the recursive formula is the following:

$$C_n = \begin{Bmatrix} 0 & if\ n = 0 \\ 1 & if\ n = 0 \\ \min\limits_{1 \le i \le n-1} \{R_{i,n} + C_i\} & if\ n > 1 \end{Bmatrix}$$

```
CanoeCost (R[],n){
  Let C[1…n] be an array
  for i ← 1 to n
    q = ∞
    for j ← 1 to i:
      if q > R[j,n] + C[j]
        q = R[j,n] + C[j]
   C[i] = q
}
```

b. Description and Pseudocode for reconstructing the problem.

We will modify the original CanoeCost function to include a new array P[1..n] which will keep track of the starting to point to get to trading post, i, $1 \leq i \leq n$. For example if the trading post was i = 5, and the best way to get to 5 was from an optimal route through trading post 3. P[5] = 3. Likewise, if the best way to get to trading post 3 was from trading post 2, P[3] =2.

The pringSequence() will print until the trading post, i = 1, the base case.

```
//We alter the original to include a new array, P[], to keep track of starting points
CanoeCost(R[],n)
Let P[1…n] be an array
Let C[1..n] be an array
for i ← 1 to n
    stop = 0
    q = ∞
    for j ← 1 to i:
      if q > R[j,n] + C[i]
        q = R[j,n] + C[j]
        stop = j
C[i] = q
P[i] = stop
}

//Print Sequence Function, it will return a string containing the sequence
PrintSequence(P[], n){
//Base Case
  if n == 1:
    return ""

  return str(P[start]) + PrintSequence(P[], P[n])
}
```

c. Recurrence equation for algorithms:
Analysis:
If we examine the outer loop $i \leftarrow 1\ to\ n, and\ the\ inner\ loop, j \leftarrow 1\ to\ i$:
Suppose n =3
$when\ i\ is\ equal\ to$: 1 2 3
$inner\ loop\ j\ exeuctes$: 1 + 2 + 3
Inner loop executes a total of 6 times

If n = 5
$when\ i\ is\ equal\ to$: 1 2 3 4 5
$inner\ loop\ j\ executes$: 1 + 2 + 3 + 4 + 5
Inner loop executes 15 times

We can see that this is an arithmetic series. The recurrence relation for the CanoeCost Function is:

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} & if\ n > 1 \end{cases} \therefore T(n) = \theta(n^2)$$

For the print sequence, the worst case would be if a canoe was dropped off and rented at each trading post, i.e. there was a stop at each post. In this scenario the function would iterate over the entire length of the array. The best case would be if the best route was from trading post 1, in this case the time would be constant. The recurrence for PrintSequence():

$$T(n) = \theta(n)$$

Problem 2. Shopping Spree

The overall problem that we are trying to solve for question 2 is what is the max profit. In order to determine this we need to determine the set items that each family member can take, taking into consideration the weight of the items and the weight bearing capacity of the family member.

We need to solve a "knapsack" problem for each family member in the test case and then sum up their total profits

Pseudocode for the knapsack()

```
Knapsack(n,w){
  Let n be the number of items
  Let w be the weight capacity
  Let w_i be the weight of item at position 1 ≤ i ≤ n
  Let p_i be the profit of item at position 1 ≤ i ≤ n
  Let C[1…n][1…w] be an array

  for i ← 1 to n:
    for j ← 1 to w:

  //compare item weight at i w_i
      if w_i > w:  //We cannot take the item
        C[i][j] = C[i − 1][j] //We take the item from the top row

  //if item weight is less that w, than we can take the item
      else:

  //if it is more profitable to take item at i plus another item that fits in the remaining weight
        if p_i + C[i − 1][w − w_i] > C[i − 1][w]:
          C[i][j] = p_i + C[i − 1][w − w_i]



        else:
  //if it is more profitable to take previously calculated set of items from the top row at w
          C[i][j] = C[i − 1][w]

  //Return the max calculated profit
  Return C[n][w]

}
```

The running time of the knapsack function is the time it takes to fill out the matrix of dimension n x w:

$$T(n) = \theta(nw)$$

The knapsack function will be used to calculate the total max profit for a given number of family members, F. We will define a function called runTest()

Pseudo Code for runTest():

```
runTest(F,n){
  Let F be the size of the family
  Let f_i be the weight capacity for a given member, 1 ≤ i ≤ F
  Let n be the number of items

  maxprofit = 0

  for i ← 1 to length(F):
    maxprofit = knapsack(n, f_i)
```

The running time for the runTest() will be the time F multiplied by the running time of the knapsack function:

$$T(n) = \theta(Fnw)$$

In order to determine the items taken for each family member, we can "walk back" the final solution of each members knapsack() function. We will define a function called takenItems():

Pseudo Code for takenItems():

```
takenItems(C[], n, w){
  Let C[1..n][1..w] be the array that was solved for in the knapsack() function for a member
  Let w be the weight capacity for a given member
  Let n be the number of items
  Let w_i be the weight of item at position 1 ≤ i ≤ n
  Let p_i be the profit of item at position 1 ≤ i ≤ n

  //Base Case
  if n == 0 or w == 0 or C[n][w] == 0:
    Return

  //Compare value at current row to one above
  If C[n][w] > C[n-1][w]    //Item was taken from row n
    Return n + takenItems(C[], n-1, w - w_n)
  //Item at row n was not taken, item was taken from row above
  Else:
    Return taeknItems(C[], n-1, w)
}
```

In the worst case an item was taken at each row, in this instance running time will be O(n), in the best case, if no items were taken, running time will be constant, c.

$$T(n) = \theta(n)$$

b. Given a test Case of n items, a family size of F, and a max weight capacity of M_i $1 \leq i \leq F$ Using the recurrence relations discussed in section a. If we assume $M = \max\{M_i, 1 \leq i \leq F\}$

$$T(n) = O(FnM)$$