1a)

```
CanoeCost(n):

currentCost = inf // current cost

CostAtEachPost[n] // array to store

For post in range n: //start at 0 till N

        if R[post,post+1] < currentCost

                costAtEachPost[post] = R[post,post+1]

                CurrentCost = R[post,post+1]

        costAtEachPost[post] = CurrentCost

return LowestCost
```

because we need a canoe to float down the river, we start of with the first post of the route and have that be the lowest cost to rent the boat. As you float down the river you are using the lowest-cost boat that you found, UNTIL you find the next cheapest one. Once you find one that is cheaper than the one you rented, you stop a that location and you get on the next cheaper boat. You repeat that until you reach the last post and end up with the lowest cost canoe.

Example:

Posts cost:	7	7	3	6	2	1	3	6	1
Lowest cost:	7	7	3	3	2	1	1	1	1

b) Yes, we can determine the canoes of the trading posts rented by determining price changes within the results.

c) Regardless of the sequence, we are required to look at each post once, and determine a fixed value of operations because of the determined decision.

$$T(N) = T(N) + O(1)$$

$$O(N)$$

2 a)

Get case info

KeyPair = [Items] [capacity the best person can carry] // store info already calculated!

Shopping (capacity, items)

If [items][capacity] has already been executed:

    Return keypair[items][capacity]

If items == 0 or capacity == 0 (base case):

    return empty set

else if cannot carry weight of item AKA capacity < item weight

    move on to next item

else:

    tmp1 = shopping (capacity, items -1,) // iterate through each possibility

    tmp2 = shopping (capacity – weight, items -1) compare to actually getting the item

    store results into KeyPair

    compare tmp1 tmp2 and return one with the highest value

the idea is to re-use some of the operations already calculated to save time and resources. Because of this we make sure that we have not already calculated the operation. If not we check the base cases and then check to see if we can actually carry the item. Finally, since we can carry the item , and the item, capacity pair was never calculated, we recursively call each item, following the function above and storing the value for future re-use.

b) theoretically, If N items are given and the family is size N, that a family member can carry at most M1, then the worst case is that the family touches every combination of N*M1, meaning we will need to do the actual calculation N*M1 times. Originally, doing this every time would have caused this program to run 2^n times.