My implementation is comprised of two main parts. First, the items we can buy are sorted by price. Then, we consider the cheapest and most expensive items in the list. If this pair of items costs more than we can afford with our gift card, then we point to the second most expensive item instead and check again. If the total cost of the two items under consideration is less than the value of the gift card, then we point to the second cheapest item to see if we can get more value. We continue this process until the two items under consideration meet somewhere in the middle. We keep track of the best value we saw along the way.

The big O of sorting the items is nlogn, and the iteration of the items is linear, so our total runtime complexity is O(nlogn). Holding a sorted array of the prices gives us a linear storage complexity O(n).

Bonus Question: What if we want to buy exactly three items?

Our pair of iterators from cheapest and most expensive will no longer work with three items, so we need a new idea. The naive approach is to run three loops and check each item against each other permutation of items, which would have a runtime complexity of $O(n^3)$. To improve upon that, we can use some memoization. Will we save the cost of every possible pair of items into a binary search tree. The runtime complexity of this is $O(n^2 \log n)$. We need to iterate through every pair of items in $O(n^2)$ time plus insert them into a tree, which adds a factor of O(logn). The storage complexity is $O(n^2)$ since we have every possible pair stored. Next, we will iterate through the list of items again in linear time. During this iteration, we find the entry in our tree that is less than or equal to the value of the giftCard minus the cost of the item under consideration. This process will be O(nlogn) since we are doing a O(logn) lookup in the BST once for every item. The highest value we saw along the way is our result.

We could use a similar approach for the two-item scenario as well.