Assignment #4

Nausher Rao (190906250) & Will Roberts (191023880)

Question 1

Explanation

The proof for the coin change problem doesn't work for the U.S post office denominations. This is because, in the case of the regular U.S currency, the algorithm starts with the greediest choice available. This is a completely optimal solution when using regular U.S currency, but in the case of postal denominations, there is always an optimal solution that differs from the greediest solution. This can be demonstrated from the examples and general rules below.

Example

For calculating the denomination make-up for 140 cents:

- For U.S currency, we can create 140 cents using 100 + 25 + 10 + 5 - this is also the optimal solution, as it only uses denominations directly less than the denomination above. Each n'th iteration uses the $n - 1$'th denomination.

- For U.S postal denominations, the greedy solution (using the cashier's algorithm) would give us 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1. This differs from the optimal non-greedy solution, this being 70 + 70.

This happens due to the fact that 100 is less than double the next smallest denomination. As a general rule, the n'th denomination has to be greater than a combination or factor of the $n - 1$'th denominations:

$$f(n) \geq k \cdot f(n - 1);\ n \geq 1,\ k \in \mathbb{N}$$

OR

$$f(n) \geq k_{n-1} \cdot f(n - 1) + k_{n-2} \cdot f(n - 2) + \ldots + k_1 \cdot f(1);\ n \geq 1,\ k_i \in \mathbb{N}$$

Where f(n) is a function that returns the value of the n'th denomination.

Question 2

Proof of Optimal Substructure

⇒ For the “Scheduling to Minimize Lateness” problem, we use an algorithm that follows the following rules:

- At the start, schedule a job for time $t = 0$.
- For jobs that have the same starting time, find the job with the least time for completion.
- Set t equal to the start time of the job + the time to complete the job.
- Go back to the start until we have reached the final time t_n for n jobs.

⇒ This algorithm utilizes the fact that at each iteration of the algorithm, the most optimal solution is used - each substructure is fully optimized.

⇒ We can use this fact to see that overall, the algorithm itself will always be optimal as each substructure is also optimal - each job selected will have minimum lateness.

Question 3

Explanation

This problem stated in Question 2 can be solved using a dynamic programming solution because the problem has both optimal substructure and overlapping subproblems.

Optimal substructure

The solution to the problem above will have an optimal substructure as the subproblem for $n - 1$ tasks will be the optimal solution, and for all prior task placement, will be optimal.

Overlapping subproblems

The solution will also have overlapping subproblems as the solution to the placement of the next task will be basses on the finishing time of the next and all other tasks before therefore the solution will have overlapping subproblems.

Because the solution to the problem will have both Optimal substructure and Overlapping subproblems therefore code can be written dynamically as dynamic programming requires the solution to follow both of these cases, which it does.

Question 4

Pseudocode

```
def main():
    p = getPositiveAmount();
    n = getNegativeAmount()
    numbers = getNumbersList();

    # sort the numbers in ascending order.
    numbers.sort();

    sum = 0;
    result = [];

    for x in numbers:
        if(n > 0):
            sum = sum - x;
            n = n - 1;

            # record that this number was a negative.
            result.add("-");
            result.add(x);

        else:
            sum = sum + x;
            p = p - 1;

            # record that this number was a positive.
            result.add("+");
            result.add(x);

    return sum, result;
```

Proof of optimality

The code will work by first sorting the given values from smallest to largest. Next, for every negative sign, the program will assign it to the next smallest available number, which will either be adding the next biggest negative value or subtracting the smallest available then for every positive sign the program will assign to the next available numbers left after the negative sign portion of the code this will mean either the smallest negative values will be added or the largest available positive numbers will be added both of these steps will lead to the largest possible value.

Question 5

We feel as if this problem is not a case for the use of dynamic programming as it neither has optimal substructure or overlapping subproblems, as the optimal solution is not based on the solution before.

Pseudocode - Memoized Technique

```
# The key for this dictionary is a tuple of the sizes and the sum
lookupTable = {};

def main(packageSizes, totalSize):
    # The key for the lookup table for this pair of arguments.
    key = (packageSizes, totalSize);

    # Returns the value in the lookup table that has already been calculated.
    if(key in lookupTable):
        return lookupTable[key];

    # If there is no solution.
    elif(len(packageSizes) == 0):
        return -1;

    # If the total size is a factor of the last package size, then the last package is the solution.
    elif(totalSize % packageSizes[-1] == 0):
        result = totalSize // packageSizes[-1];
        lookupTable[key] = result;

        return result;

    # The last package is not the solution → Lets go to the next last package size.
    else:
        return main(packageSizes[:-1], totalSize);
```

Time Complexity - Memoized Technique

The program will start at the last element of a sorted list and will check if m is divisible by the last item of the sorted list. The program will have found a solution. Next, the program will check if the current value is the only value in the list. If this is true, the program will return that no solution is available.

Lastly, if the the current largest size is not a solution and the list has more then one value, the program will recursively call the function with the last element of the list removed. As the program will visit every value, the worst case is that no solution is found - the worst-case run time will be $O(n)$.

Since we know that the algorithm iterates through every element in the array recursively, it iterates n times. The array size decreases to $n - 1$ each time, therefore $T(n - 1)$. With each lookup in each iteration, +1 operations are included.

Therefore, $O(n) = T(n) = T(n - 1) + 1$

Pseudocode - Bottom-Up Technique

```
def main():
    packageSizes = getPackageSizes();
    totalSize = getTotalSize();

    for packageSize in packageSizes:
        if(packageSize is a multiple of totalSize):
            We have found a solution!

            numberOfPackages = totalSize // packageSize
            packageSize = the size of each package.

            We can now quit the program since we have found a solution.


    if we reach here, we haven't found a solution!
```

Time Complexity - Bottom-Up Technique

The program will visit values one at a time until it finds a value that will work with the given m value, or it reaches the end of the list. The best case of this program will be $O(1)$ if the first largest value in the list is a solution. The worst case will be when there is no solution as the program will iterate through whole list, so the run time then will be $O(n)$.