Jared Raiola HW2

Q1]

a) T(i) is a table containing entries that are the sum of all values up to and including that index.

b) T(i) = T(i-1) + S(i)

Subproblem:

↳ Sum at each index of list S

c) function contiguousSubsequence(list S):

```
    if length of S == 0:
        return [] and 0
    if length of S == 1:
        return [S[0]] and S[0]
    list S ← size n
    L[0] = S[0]
    maxValue = S[0]
    maxIndex = 0
    minValue = S[0]
    minIndex = -1 ← We have no true min yet
    for i from 1 to n-1:
        L[i] = L[i-1] + S[i]
        if L[i] > maxValue:
            maxValue = L[i]
            maxIndex = i
        else if L[i] < minValue:
            minValue = L[i]
            minIndex = i
```

```
end for
if maxIndex != 0:
    if minIndex != -1:
        finalValue = L[maxIndex] - L[minIndex]
    else:
        finalValue = L[maxIndex]
        minIndex++
    list C ← size maxIndex - minIndex
    for i from minIndex to maxIndex + 1:
        C[i] = S[i]
    return C and finalValue
```

d) Runtime: O(n)

↳ Justification:
We loop through the subsequence once giving us linear time. All other calculations are done in less than n time, giving us a constant value that can be ignored.

↳ Why this works:
The difference between the lowest sum and highest sum when traversing in one direction with the constraint of being a CONTIGUOUS subsequence will always be the greatest sum. This occurs when max sum is found further into the subsequence than the min sum, otherwise the min sum is the beginning.

Jared Raiola HW2

Q2

a) T(i,j) is a table of size n by m where each entry is the length of the current common substring.

b) T(i,j) = T(i-1, j-1) + 1
Subproblems:
↳ Checking each occurrence of a character and if the character before it also existed in the spot before.

c)
```
def LCS(string x, string y):
    list D ← 2 dimensions, size n by m
    length = 0
    for i from 0 to n-1: ← length of x
        for j from 0 to m-1: ← length of y
            if x[i] == y[j]: ← match
                if i == 0 or j == 0: ← first row or column
                    D[i][j] = 1
                else:
                    D[i][j] = D[i-1][j-1] + 1
                if D[i][j] > length:
                    length = D[i][j]
            else:
                D[i][j] = 0
    return length
```

d) Runtime: O(nm)

b) Justification:

Since we dynamically update the lookup table with the current substring length, we can do this in one pass of string x with one pass of string y nested in it.

b) Why this works:

The lookup table is only incremented every time two characters match. It is then only incremented by 1. This means that if the previous character that was looked at in the outer loop had a match, and the lookup of T(i-1, j-1) (up 1, left 1) returns a value, we can accurately tell if they were in order if that value is greater than or equal to 1. If it isn't, then we only currently have a substring of 1.

Jared Raiola HW2

Q3]

a) T(i) is a table of size V+1, as we treat all values from zero to V ($0 \leq q \leq V$) as valid, where entries contain the number of coins in the highest denominations that can go into that index perfectly. If perfect change cannot be made, the entry will be -1, as it is an invalid entry.

b) $T(i) = S(q - x_i) + 1$; if $S(q - x_i) != -1$
where $0 \leq q \leq V$ (Knapsack problem).
Subproblems:
↳ Treat as a knapsack problem where we check every value from 0 to V for validity
↳ Sort list of coin denominations, X
↳ Check K at the end

c) function makeChange(x, v, k):
    mergeSort(x) ← Order x lowest -> highest
    list N ← size V+1
    N[0] = 0 ← It takes 0 coins to make 0 value
    for q from 0 to V:
        for i from 0 to n-1: ← $x_0 \ldots x_n$
            if $q \geq x[i]$ and $N[q - x[i]] != -1$:
                $N[q] = N[q - x[i]] + 1$
    return $N[v] \leq k$

d) Runtime: $O(n \log n + nV)$

↳ Justification:

We are looking through all possible coin denominations ($X_1 \ldots X_n$) for every value from 0 to V from the lowest coin denomination to the highest denomination due to merge sort.

↳ Why this works:

If a lower value can be satisfied by the coin denominations, and that value is any denomination in $X_1 \ldots X_n$ away from the value of V or another lesser value, the coins can sum to that value.

We will always get the lowest number of coins to check against K as mergesort forces the denominations to be in order, prioritizing the higher valued coins.

E.g.

makeChange([5,10], 15, 2)

[0, -1, -1, -1, -1, 1, -1, -1, -1, -1, 2 ... ...2]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

2 with 5 input on first pass gets rewritten as 1 when 10 is checked

Jared Raiola HW2

Q4)

a) T(i) is a table of size n, where entries are the current highest possible sum of non-consecutive house values.

b) $T(i) = P(i) + T(i-2)$
if $P(i) + T(i-2) > T(i-1)$
else $T(i) = T(i-1)$
Subproblems:
b) Current house value plus house$_{i-2}$ value vs house$_{i-1}$ value

c) function steal(P): $\leftarrow P_1 \dots P_n$
if n == 1:
return P[0]
if n == 2:
if P[0] > P[1]:
return P[0]
else:
return P[1]
list M ← size n & list of sums
M[0] = P[0]
if P[0] > P[1]:
M[1] = P[0]
else:
M[1] = P[1]
end else

```
for i from 2 to n-1
    if P[i] + M[i-2] > M[i-1]:
        M[i] = P[i] + M[i-2]
    else:
        M[i] = M[i-1]
return M[n-1]
```

d) Runtime: O(n)

↳ Justification:
We only loop through the houses once, giving it linear time.

↳ Why this works:
We are always taking the highest sum of houses, because the next highest sum will be stored at i-2 so we can make the comparison between adding on to the i-2 sum vs keeping the highest sum. We only truly need to worry about blocks of 4 houses at a time, because that creates awkward adjacent selection. Blocks of houses of 3 and under allow for the highest two or one to be selected, so we choose to deal with this relation.