CSCI 2300 — Algo
Homework 8
Hayden Fuller

- **QA** A: Textbook questions:
  - **1) 6.1** A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is $5, 15, -30, 10, -5, 40, 10$, then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task: Input: A list of numbers, $a_1, a_2, ..., a_n$. Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero). For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55. (Hint: For each $j \in \{1, 2, ..., n\}$, consider contiguous subsequences ending exactly at position j .)
    Subproblem: I don't keep a full array since moost of the values imediately become irellivent, but endHereMax during any given loop will show the maximum sequence you can have that ends at that index. curMax tracks those maxes as the program runs and reports the max of that (along with start and end indexes)
    Recurrence: uses the max that ends at the previous index to find the max that ends at this index
    Algorithm:

```
def MaxSub(a,n):
    curMax = -infinity
    endHereMax = 0
    start=0
    end=0
    s=0

    for i in range(0,n): #for each index
        endHereMax+=a[i] #max from start to here is previous plus this spot
        if endHereMax > curMax: #if a new max is found, set that is current max to
                                                  #compare and record start and end points
            curMax = endHereMax
            start = s
            end = i
        if endHereMax < 0: #if current sum is negative, the max is just 0 for the
                               #empty array, so reset to 0 and use a new starting point
            endHereMax = 0
            s=i+1

    print(f'Maximum is {curMax} from index {start} to {end}')
```

    Correctness: since we have the max that ends on the previous index, the max that ends on this index will either be this index plus the previous plus this one, or it will restart to zero.
    Running time: just one for loop of n gives us O(n)
  - **2) 6.4** You are given a string of n characters $s[1...n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like

"itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict(·): for any string w, dict(w)={true if w is a valid word, false otherwise.

(a) Give a dynamic programming algorithm that determines whether the string s[·] can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to dict take unit time.
Subproblem: Can a word be broken off at this point?
Recurrence: We use the starting point of a future word to make sure they all line up without gaps
Algorithm:

```
def fixable(s):
    n = len(s)
    b = [False] * (n + 1) #breaks, can a word be broken off here
    b[n] = True #end counts as a word break
    for i in range(n - 1, -1, -1): #starting at the end and going back
        for j in range(i+1,n-1): #for each possible word starting at index i
            if b[i + len(s[i,j])]: #only the words that end where
                                           #another word started
                if dict(s[i,j]): #if this word is in the dictionary,
                                         #break it off here
                    b[i] = True
                    break
    return b[0] #if the last (starting) word can be split off, then that
                      #means the whole thing can be split into words
```

Correctness: Base: a word can end at the end of the sentence. Inductive: if a word can be made from index I to the start of another known word, then you can break that index off as the start of a word that another word can end at.
Running time: for loop of n within another for loop of n gives us $O(n^2)$

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words
Subproblem: Can a word be broken off at this point and if so, where does it end?
Recurrence: We use the starting point of a future word to make sure they all line up without gaps
Algorithm:

```
def fix(s):
    output=[]
    n = len(s)
    b = [-1] * (n + 1) #breaks, can a word be broken off here, and if so, where
                               #does that word end?
    b[n] = n #end counts as a word break that ends at itself
    for i in range(n - 1, -1, -1): #starting at the end and going back
        for j in range(i+1,n-1): #for each possible word starting at index i
            if b[i + len(s[i,j])]: #only the words that end where another
                                         #word started
                if dict(s[i,j]): #if this word is in the dictionary, break it
                                         #off here, and track what index this word
                                         #goes to
```

```
                    b[i] = i + len(s[i,j])
                    break
        if b[0]: #if the last (starting) word can be split off, then that means the
                    #whole thing can be split into words
            start=0 #start of word
            end=b[0] #end of word
            while start < n: #until start reaches the end
                output.append(s[start,end])
                start=end #start of next word is the end of this one
                end=b[end] #b[end] is the end of the next word
            return output #output is an array of all the words It's possible for
                        #there to be multiple solutions, but this will only be one.
```

Correctness: Same breaks and mainly the same reasoning as the last algorithm, this just keeps track of them, and then goes from word break to word break using dynamic programming (since we stored where the end of each word is at it's start) for the output
Running time: now with an O(n) output added to the end, but otherwise same as lasts time, for loop of n within another for loop of n gives us $O(n^2)$

– **3) 6.17** Given an unlimited supply of coins of denominations $x_1, x_2, ..., x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem. Input: $x_1, ..., x_n$; v. Question: Is it possible to make change for v using coins of denominations $x_1, ..., x_n$?
Subproblem: How many coins does it take to make this value? Note: This could more easily be done with True/False, but I read the next problem and this works
Recurrence: look at previous values, because if you have a c cent coin and you can make value v-c, you can make v by adding the c
Algorithm:

```
def coinProb(v,coins):
    n=len(coins)
    array=[infinity for i in range(v+1)]
    array[0]=0
    for i in range (1,v+1): #for each value up to v
        for j in range(n): #check each coin
            if (coins[j] <=i): #if coin is less than the current value
                                 you're looking for
                sub = array[i-coins[j]] #if you can reach the current value you're
                                         looking for minus one coin, you can just
                                         add that coin to get this value
                if (sub != infinity and sub+1<array[i]):
                    array[i]=sub+1
    return array[v] < infinity #if change can be made, True
```

Correctness: base: 0 can be made with 0 coins. inductance: if you have a coin valued c and can make value v-c, you can make value v by adding c
Running time: for loop of n in a for loop of v, giving O(nv)

- **4) 6.19** Variation of the above problem: you can only use maximun number of coins k. Input: $x_1, ..., x_n$; v. Question: Is it possible to make change for v using at most k coins of denominations $x_1, ..., x_n$?
  Subproblem: How many coins does it take to make this value? Note: same algorithm as above, just a different comparison at the end
  Recurrence: look at previous values, because if you have a c cent coin and you can make value v-c, you can make v by adding the c
  Algorithm:

```
def coinProb(v,coins,k):
    n=len(coins)
    array=[infinity for i in range(v+1)]
    array[0]=0
    for i in range (1,v+1): #for each value up to v
        for j in range(n): #check each coin
            if (coins[j] <=i): #if coin is less than the current value
                                  you're looking for
                sub = array[i-coins[j]] #if you can reach the current value you're
                                           looking for minus one coin, you can just
                                           add that coin to get this value
                if (sub != infinity and sub+1<array[i]):
                    array[i]=sub+1
    return array[v] < k #if change can be made with less than k coins, True
```

  Correctness: base: 0 can be made with 0 coins. inductance: if you have a coin valued c and can make value v-c, you can make value v by adding c. Since you check all possible coins and find the minimum, you'll know you can just add 1 to get the number of coins required for this value.
  Running time: for loop of n in a for loop of v, giving O(nv)

- **QB** Consider the "Weighted Interval Scheduling" problem discussed in the class with the following requests

```
R={ r1,r2,r3,r4,r5,r6} where
r#: (Start time Finish time Value)
r1: ( 1 3 2 )
r2: ( 2 5 4 )
r3: ( 4 5 3 )
r4: ( 2 7 7 )
r5: ( 6 8 2 )
r6: ( 6 9 1 )
```

  1. Implement an iterative DP Algorithm to find the subset of requests that has the total maximum value. (Hint: us an array M to store the values as described in the class)
     Note: this assumes requests is sorted, if it's not, quick sort.
     Subproblem: What is the max value scheduel and what requests are used to achieve it up to a certian time
     Recurrence: We use either the max of the previous request or the max of a previous non

overlapping request
Algorithm:

```
def maxScheduel(values, requests):
    n=len(requests)
    M=[0 for i in range(n)]
    subset = [[] for i in range n]
    for i in requests: #for each request
        #max at this point is either still using the max of the previous, or using
        #the max of the previous that doesn't overlap plus this new one
        maxm=0 #max M value
        maxi=0 #index of that value
        for(j in range(i)): #find what index j has max M[j] before i without
                            #overlapping
            if(range[j].finish <= range[i].start and M[j]>maxm):
                maxi=j
                maxm=M[j]
        if (M[i-1] > values[i]+M[j]): #if it's best to just use the previous request's
                                              #max and leave empty time,
                                              #steal the values from previous
            subset[i]=subset[i-1]
            M[i]=M[i-1]
        else: #better to add in this new request to the previous non overlapping
             #request, steal it's values and add to it
            subset[i]=subset[j]
            subset[i].append(i)
            M[i]=M[j]+values[i]
    return subset[n]
```

Correctness: max is either that of the previous and leaving blank time or it's adding this in and using the max that doesn't overlap
Running time: for loop of n in a for loop of n, giving O(n^2), I know it can be faster, but this is what I got.

2. Implement an algorithm and print the list of intervals in the optimal solution above by using the array M without maintaining another data structure
I'm out of time, but you can just repeat the exact same thing and make M a 2d array to store all the same information