Question 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, a1, a2, . . . , an.
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 ∈ {1, 2, . . . , n}, consider contiguous subsequences ending exactly at position j.)

Following the hint, we'll break the problem into smaller subproblems.

Let sums[i] be an array of size equal to the length of S (the input list). Sums[i] will represent the sum of the continuous subsequence of maximum sum that ends at index i in the initial list S.

Iterating through S, we can use this recursive formula to populate sums[i] with values.

$$sums[i] = \begin{pmatrix} s[i] & if\ sums[i-1] < 0 \\ sums[i-1] + S[i] & if\ sums[i-1] \geq 0 \end{pmatrix}$$

If the maximal sum up until index i is negative, then we're always better off "starting over" and making the next item in S the "start" of a new potentially maximal sequence.

If the maximal sum up until index i is not negative, then we simply append the next item in S to the end of the current subsequence.

For the base case, we can set Sums[0] equal to S[0], since the maximal subsequence ending at index 0, is just the value of the first number.

With this definition, we can obtain the maximal sum and the subsequence that generates that sum by maintaining a list of "previous" pointers and continuously updating a "max sum" variable. Here's an example algorithm given an input sequence of integers, S:

```
// If initial list is empty, return 0
If len(S)==0 {return 0}

// Initialize variables
Sums[] = [0]*len(S)
Prev[] = [0]*len(S)
Max_index = 0
Max_sum = 0
```

```
// Loop over the list S, filling in Sums
For i = 0  < len(S) {

        // Use recursive definition from above, adjusted slightly for the base case
        If i==0 or sums[i-1] < 0:
                Prev[i] = null
                Sums[i] = S[i]
        Else:
                Prev[i] = i-1
                Sums[i] = Sums[i-1] + S[i]

        // Update max_sum and max_index
        If Sums[i] > max_sum:
                Max_sum = Sums[i]
                Max_index = i
}

// Initialize an empty list to hold the ordering
Ordering = []
I = max_index
While (max_index not null) {
        Ordering.append(S[i])
        I = prev[i]
}

Ordering.reverse()
```

At the end of this algorithm, max_sum holds the sum of the maximal contiguous subsequence and ordering holds a list of items of S that makeup the maximal contiguous subsequence.

The runtime is O(n) because we loop over the entire sequence once, then then maximal subsequence sequence once.

Question 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.

Start with a boolean array d[i] equal in length to the size of the input string / character array s[i]. D[i] will hold "true" if a substring of the input string from 0 to i (inclusive) can be split into valid words. String[0…i] or D[i] is a valid sequence of words only if there is some value(s), j, where string[0…j] is a sequence of words and string[j…i] is a word. With this in mind, we can recursively fill d[i] in with values like so.

$$d[i] = \begin{pmatrix} true & if\ \forall\, j, 0 \le j \le i, d[j] = True\ AND\ string[j+1\ to\ i]\ (inclusive)\ is\ also\ a\ word \\ false & otherwise \end{pmatrix}$$

For this problem, we can say that the base case is d[0], the empty string which we'll consider a valid word for the purposes of our algorithm.

Applying to the definition to an algorithm, we get:

```
def valid_sequence(string):

  // String is the input string
  // We'll prepend a meaningless character to the string to account for the empty string base case
   string = "!" + string

   // initialize array to hold previous solutions
   d = [False]*(len(string))

   // Set the base case equal to true
   d[0] = True

   // Use the recursive formula from above
   for i in range(len(string)):
      for j in range(i):
         if (d[j]==True)and(Dict(string[j+1:i+1])==True):
            d[i]=True

   return d[len(string)-1]
```

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

We just need to adjust the algorithm to store the indices that denote the ends of words. This occurs when the if statement in our recursive definition is True. Afterwards we can use the stored indexes to substring out the words in the sequence, reverse them, then output them. Here is the new code (see comments for further explanation):

```
string = "!" + string

   d = [False]*(len(string))
   c = [0]*len(string) // this array will store indexes that denote word ends
   d[0] = True

   for i in range(len(string)):
      for j in range(i):
         if (d[j]==True)and(Dict(string[j+1:i+1])==True):
            d[i]=True
            c[i]=j // In this context, j denotes the end index of a valid string of words and the beginning of a new word.


   order = []
   i=len(string)-1 // the last word end will always be the last index of the string (if the string is a valid sequence of words)
   while i>0:
      order.append(string[c[i]+1:i+1]) // substring out a word from the previous word end to the current word end.
      i = c[i] // update I to point to the end of the previous word.

   // Reverse the list since the above code adds the words in reverse order.
   order = list(reversed(order))
   print(order)
   return d[len(string)-1]
```

Question 6.17

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, 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: $x_1, \ldots, x_n$; v.
Question: Is it possible to make change for v using coins of denominations $x_1, \ldots, x_n$?

Define an array of size v, C[i] that holds true if it makes change for value i.

We can come up with a recursive definition for C[i] like so:

$$C[i] = (C[i - x_j] \quad if\ x_j \leq i\ for\ 0 < j < n)$$

The logic is that if we know we can make change for value v, then if add a coin of value x to v, we now know we have the ability to make change for value v+x. We do this process on all values from 1 to v with all possible coin denominations.

For this scenario, the base case is C[0]. We'll set this value to true, because if we have zero coins in our change pile so far, then we can always add a coin of value x to make change for that same value x.

Here's the pseudocode implementation:

```
# This method takes X, the array of coin denominations, and v the value to be equaled by the set of coins
def make_change(X,v):

    # Initialize the array C
    C = [False]*(v+1)

    # Handle the base case
    C[0] = True

    # Loop up until the max value
    # Move range up by 1 to handle the base case
    for i in range(1,v+1):
        for j in range(len(X)):

            # Implement recursive function as described above
            if X[j] <= i:
                C[i] = C[i] or C[i-X[j]] # Add an OR so that if any coin combination works, then C[i] is set to true
            else:
                C[i] = False

    #Return last entry which gives us the final answer
    return C[v]
```

Question 6.19

Here is yet another variation on the change-making problem (Exercise 6.17).

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v using at most k coins; that is, we wish to find a set of ≤ k coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 and k = 6, then we can make change for 55 but not for 65. Give an efficient dynamic-programming algorithm for the following problem.

Input: $x_1, \ldots, x_n$; k; v.
Question: Is it possible to make change for v using at most k coins, of denominations $x_1, \ldots, x_n$?

Let M[i] be an array that will be storing the minimum number of coins required for value i.

Here is a recursive definition for M

$$M[i] = \big(M[i] = M[i - X[j]]) \quad if\ X[j] \leq\ i\ for\ all\ 0 < j < i\ AND\ M\big[i - X[j]\big] < M[i]\big)$$

We can simply extend the functionality of the formula from the previous question so that values in M only get updated if the new combination of coins uses less coins that a previous entry in M[i].

Just as before, we'll set the base case to M[0] = 0, because 0 coins are used to make change for value 0.

```
# Function to calculate if change can be made with coins in X for value V with at most k coins
def minCons(X,V,k):

    # Initialize array
    M = [None]*(V+1)

    # Set base case
    M[0] = 0

    # Apply recursive formula, slightly altered for weird indexing that happens because of the base case
    for i in range(1,V):
        for j in range(len(X)):
            if X[j] <= i and M[i-X[j]]+1<M[i] and M[i-X[j]] != None:
                M[i] = M[i-X[j]]+1

    # Check to see if a small enough amount of coins was found
    if M[V]<=k:
        return True
    else:
        return False
```