Answer the questions below through the Udacity site. Note that for some questions it is not sufficient to simply pass the tests on the site. Your code must have a certain running time, and in some cases you must also answer additional questions in the comments of your code.

1. Suppose there is a procedure f that decides a language A in $O(g(n))$ time, where n is the length of the input string. Use dynamic programming (not memoization) to create an algorithm that decides A^* in $O(n^2 g(n))$ time. Implement your strategy in python here.

Solution:

```
def inAstar(x, inA):
    """ Returns true if x is in A^* and false otherwise.
    The input inA should be assumed to be a function
    that takes are string as input and return True or False."""

    n = len(x)

    #This array will contain the starting index
    #of the predecessor string in A in the sequence.
    pred = np.zeros(n+1)

    #The empty string is always in A*
    pred[0] = 0

    for i in range(1, n+1):
        pred[i] = -1
        for j in range(i):
            if pred[j] >=0 and inA(x[j:i]):
                pred[i] = j
                break

    return pred[n] >= 0
```

2. Consider an arbitrary string $X = x_0 \ldots x_{n-1}$. A *subsequence* of X is a string of the form $x_{i_1} x_{i_2} \ldots x_{i_k}$ where $1 \leq i_1 < \ldots < i_k < n$. For example 'ape' is a subsequence of 'apple.' A string is *palindromic* if it is equal to its own reverse (i.e. the string is the same whether read backwards or forwards).

We will use dynamic programming to find the length of the longest palindromic subsequence of X. For $i \leq j,$ let $L(i,j)$ be the length of the longest palindromic

subsequence of the substring $x[i:j]$ (the substring going from the ith character to the jth character of x, not including the jth character).

a. Derive a recurrence for L(i, j), remembering to include the base case?
b. Why is your recurrence correct?
c. Write a dynamic programming algorithm based on the recurrence for L(i, j) in python. Use backtracking to construct a longest palindromic subsequence.
d. What is its running time?

Solution:

a. For any substring $x[i:j]$ there are three possibilities for the longest palindromic subsequence.
   i. It can include the first and last characters on either end if those characters are equal. In this case, $L(i,j) = L(i+1,j-1)+2$.
   ii. It can exclude the last character. In this case, $L(i,j) = L(i,j-1)$.
   iii. Or it can exclude the first character. In this case, $L(i,j) = L(i+1,j)$.

   (Excluding both is included in the previous two possibilities). Therefore, we take the maximum of these three possibilities to find the maximum length of a palindromic subsequence, i.e. $L(i,j) = \max\{(L(i+1,j-1)+2)[x[i] == x[j-1]], L(i,j-1), L(i+1,j)\}$.

   The base cases are $L(i,i) = 0$ and $L(i,i+1) = 1$, for all relevant i. The empty string has a maximum palindromic subsequence length of zero, and any single symbol string has a maximum palindromic subsequence of length of 1.

b. Consider some longest palindromic subsequence of $x[i:j]$. If $x[i] = x[j-1]$, then this is sequence cannot be more than two longer than the longest palindromic subsequence of $x[i+1:j-1]$. On the other hand, if it does they do not match, then it is the same length as either a palindromic subsequence of $x[i+1:j]$ or of $x[i:j-1]$. Since we are taking a maximum over the quantities on the right hand side, this shows the $\leq$ direction.

   In the other direction, we observe that any palindromic subsequence of $x[i+1:j-1]$, of $x[i+1:j]$, or of $x[i:j-1]$ is also a palindromic sequence of $x[i:j]$. In the $x[i+1:j-1]$ case, then there is a palindromic sequence exactly two larger. This shows the $\geq$ direction.

c. Here is my code

```
def lps(s):
    """Returns a longest palindromic subsequence of the string s."""
    n = len(s)
    L = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
```

```
    L[i][i] = 1

def best_choice(i,j):
    return max( L[i+1][j], L[i][j-1], L[i+1][j-1] + 2 if s[i]== s[j] else 0)

for d in range(1,n):
    for i in range(1,n-d):
        j = i + d
        L[i][j] = best_choice(i,j)

i = 0
j = n-1
pre = []
while j > i:
    if s[i] == s[j] and L[i+1][j-1] + 2 == best_choice(i,j):
        pre.append(s[i])
        i = i + 1
        j = j - 1
    elif L[i+1][j] == best_choice(i,j):
        i = i + 1
    else:
        j = j - 1

ans = ''.join(pre)
if j == i:
    ans = ans + s[i]

pre.reverse()

return ans + ''.join(pre)
```

d. The running time is $O(n^2)$, with filling out the L matrix being the dominant term.

3. Let $A(x) = a_0 + a_1x + \ldots + a_nx^n$ and let $B(x) = b_0 + b_1x + \ldots + b_nx^n$ n be two polynomials. Recall that $C(x) = A(B(x)) = a_0 + a_1B(x) + a_2B^2(x) + \ldots a_nB^n(x) = c_0 + c_1x + \ldots c_{n^2}x^{n^2}$. Implement an $O(n^3 \log n)$ for finding the coefficients of C at

Explain your running time analysis in your comments. It's okay if its faster than $\Omega(n^3 \log n)$.

(Bonus 10pts) Make your algorithm $O(n^3)$. Explain your running time analysis in your comments.

Solution:

```
#Strategy 1
def polycomp_conv(A,B):
    """Computes A(B(x))"""
    m = len(A)
    n = len(B)

    C = np.zeros((m-1) * (n-1) + 1, dtype=int)

    #O(m^2n log nm) overall.
    k = 1
    for i in range(0,m-1):
        C[k-1] += A[i] #O(1)
        C[:k+n-1] = np.convolve(C[:k], B) #O(nm log nm) in the last step
        k = k + n - 1
    C[-1] += A[-1]

    return C

#Strategy 2
def polycomp_fft(A,B):
    m = len(A)
    n = len(B)

    #O(mn^2) is the dominant term here.
    B = B[::-1]
    fft_b = fft(B, (m-1) * (n-1) + 1) #O(mn log mn)

    fft_ab = np.polyval(A,fft_b) #(O(mn^2))

    C = ifft(fft_ab) #O(mn log mn)
    C = C[::-1] #O(mn)

    return C.real.round().astype(int)
```

4. Given two sets A and B of integers, their sum set C is defined to be $C = \{a+b : a \in A, b \in B\}$. For each $c \in C$, let $m(c)$ denote the number of ways in which c can be obtained, i.e., $m(c) = |\{(a,b) : a \in A, b \in B, a+b = c\}|$. For example,

if $A = \{1,2,3\}$ and $B = \{0,1,4\}$, then their sum set is $C = \{1,2,5,3,6,4,7\}$ and $m(1) = 1$, $m(2) = 2$, $m(3) = 2$, $m(4) = 1$, $m(5) = 1$, $m(6) = 1$, and $m(7)$ = 1.

a. For two arbitrary sets A and B of integers, what is the worst-case complexity of computing their sum set?
b. Suppose now that A, B consist of integers in the range $[0, n)$. Give an upper bound on the number of distinct elements in their sum set.
c. With the above assumption, give an $O(n \log n)$ to find $(c, m(c))$ for each $c \in C$.

Answer the above questions in the comments of your submission to Udacity at
y

Solution:

a. All sums $a + b$ could be distinct, so that even writing out the numbers would take $\Omega(|A||B|)$ time. Simply iterating over all-pairs achieves this complexity.
b. In this case, the least possible element of C is 0 and the greatest is $2n - 2$. Thus, there are at most $2n - 1$ possible values.
c. We can represent each set A and B as a polynomial consisting of 0-1 coefficients. Let $A(x) = \sum_{a \in A} x^a$ and $B(x) = \sum_{b \in B} x^b$. Then $A(x)B(x) = \sum_{c \in C} m(c)x^c$. Therefore, to compute m(c) we need only convolve the coefficient representations of the nth order polynomials A and B. This take time $O(n \log n)$ using the FFT.

```
def sumset(n,A,B):
  a = np.zeros(n, dtype = int)
  b = np.zeros(n, dtype = int)

  for x in A:
    a[x] = 1
  for x in B:
    b[x] = 1

  c = np.convolve(a,b)

  return [ (i,c[i]) for i in range(len(c)) if c[i] > 0]
```