Question 1 (Longest Increasing Subsequence):

1. We can adjust the solution presented in class for the LIS case.   
   We need to make sure that the odd –even repetition holds.

Now using this recursion function we can complete the DP algorithm. The input is a sequence An of size n, L is a vector of size n an L1 = 1. For i from 2 to n we update according to the recursion function.

1. Time complexity - compared to of the exhaustive solution. Space complexity -   
   compared to of the exhaustive solution

Question 3 (Language Model):

1. 1. We will know introduce this problem as a FHDP whilst K is the number of stages and in each stage we will represent the letter taken for the most probable word. We will now define the state space, action space, transition function and multiplicative cost function.

**State space -**   
**Action space -**

**Transition Function -**

**Multiplicative cost function -**  in this way when we try to minimize the cost we equivalently try to maximize the probability.  
2.Time complexity is given by O(K\*|A|\*|S|)=O(9K)=O(K)

Space complexity is the total number of states which is 3(K-1) + 2 = O(K)

3. analogous problem with additive cost function is one that applies the log function to the multiplicative cost function mention in b.1.

4. the tradeoff is between computation error and memory size. When K is big we will prefer using log in order to deal with smaller numbers and when K is small we will prefer using multiplication because log will have some numerical errors.

Code:

import numpy as np  
  
  
def mostProbableWord(P, letters, K):  
 costs = -np.log(P)  
 n\_letters = len(letters)  
 cumulative\_cost = np.zeros((n\_letters, K + 1))  
 actions = np.zeros((n\_letters, K + 1))  
 for stage in reversed(range(0, K)):  
 for state in range(n\_letters - 1): # avoid last letter  
 if stage == K - 1: # final stage - only last letter is legal  
 cumulative\_cost[state, stage] = cumulative\_cost[n\_letters - 1, stage + 1] + costs[state, n\_letters - 1]  
 actions[state, stage] = n\_letters - 1  
 else:  
 possible = range(n\_letters - 1) # intermediate stage - last letter is illegal  
 cumulative\_cost[state, stage] = np.min(cumulative\_cost[possible, stage + 1] + costs[state, possible])  
 actions[state, stage] = np.argmin(cumulative\_cost[possible, stage + 1] + costs[state, possible])  
 # Now we have the chosen actions for each state in each stage, use this to find optimal path:  
 actions = actions.astype(int)  
 optimal\_path = np.zeros(K, dtype=int)  
 for i in range(1, K):  
 optimal\_path[i] = actions[optimal\_path[i - 1], i - 1]  
 word = letters[optimal\_path]  
 word[0] = word[0].upper()  
 return ''.join(word)  
  
  
# Q3:  
P = np.array([[0.1, 0.325, 0.25, 0.325], [0.4, 0, 0.4, 0.2], [0.2, 0.2, 0.2, 0.4], [1, 0, 0, 0]])  
# first letter in letters is the must be first letter and last letter in letters is the end token  
letters = np.array(['b', 'k', 'o', '-'])  
probable\_word = mostProbableWord(P, letters, 5)  
print('Q3: length K = ', 5, probable\_word)

Result: for K =5 we get that ‘Bkbko’ is the most probable word