Dynamic Programming

1. [***De-merging***]
If two sequences $a_1, a_2,..., a_m$ and $b_1, b_2,..., b_n$ are interleaved, we say that the resulting sequence $c_1, c_2,..., c_{m+n}$ is a shuffle of the first two. For example,

DCCDBDADCACDBACB

is a shuffle of DCBDAACBB and CDDCDAC since it can be obtained by interleaving those two sequences in this way:

```
DC    BDA     AC   B     B
   CD      DC    D   AC
```

You are to give a dynamic programming algorithm for determining whether or not a given sequence is a shuffle of two other given sequences. Your algorithm is to run in time O(mn), where m, n and m + n are the lengths of the three sequences. You should carefully describe each step of the process, e.g. this should include certain definitions, the principal recurrence relations, the table setup, the order in which the table contents are computed, and the computation storage window utilized.

From a simple perspective, the problem does not seem very difficult. In order to determine whether a given sequence is a shuffle of two other given sequences, one pass must be made through the merged sequence of size *n* + *m* and at each element in the sequence, a comparison must be made against the current element in both the root sequences. Because each element generates two comparisons, one in the sequence of length *n* and one in the sequence of length *m*, and because each of these comparisons are, in actuality, recursive calls, the overall complexity of this solution would be $O(2^n)$.

This solution has a complexity of $O(2^n)$ because it redundantly completes sub-problems at each element. In order to solve this using a Dynamic Programming algorithm, the result of each sub-problem must be saved so it can be referenced later, resulting in a more efficient algorithm, one that will hopefully run in O(*mn*) time.

One example of a dynamic programming algorithm to solve this problem involves the use of a table to store the progress of the shuffle states. Each state is saved as a boolean value of either True or False and is determined using specific values of the two root strings as well as the state values in the preceding cells above and to the left of the current cell.

The algorithm begins by creating the memoization table, table *T*, of size *m* x *n*. Next, the value at *T*[0, 0] is initialized to TRUE while the remaining values in the table are initialized to FALSE. At this point, one pass is made through both the root strings comparing each to the merged string beginning at the first character of each and iterating until the strings are no longer identical *OR* until the end of each substring is reached:

```
i, j = 1

WHILE A[i] == C[i] && i <= len(A)
     T[i, 0] = TRUE

WHILE B[j] == C[j] && j <= len(B)
     T[0, j] = TRUE
```

The results of this process for the given sequences is shown in the table below:

			Sequence B								
			D	**C**	**B**	**D**	**A**	**A**	**C**	**B**	**B**
		1	1	1	0	0	0	0	0	0	0
Sequence A	**C**	0									
	D	0									
	D	0									
	C	0									
	D	0									
	A	0									
	C	0									

At this point, the algorithm begins iterating through rows, *i*, and then through columns, *j*. In order to make this explanation simpler, assume the root sequences *A* and *B* and the merged sequence *C* are stored in arrays beginning with index 1 (e.g., *A*[1] = 'C', *B*[1] = 'D', *C*[1] = 'D').

While iterating through rows and then columns of a given row, the value of any given cell, *T*[*i*, *j*], is calculated using the following boolean operation:

```
( A[i] == C[i + j] AND T[i-1, j] == TRUE )
OR
( B[j] == C[i + j] AND T[i, j-1] == TRUE )
```

In other words, for any given state, cell *T*[*i*, *j*], the value is TRUE when either the current value for A is equal to C[*i* + *j*] and the cell above it is TRUE, *OR*, the when the current value for B is equal to *C[i + j]* and the cell to the left of it is TRUE. Given this logic, if the cell above and the cell to the left are both FALSE, the current cell must have a value of FALSE. If, however, both cells are TRUE, the current cell is not necessarily TRUE.

This algorithm continues until the table is completely filled, which occurs when $T[m, n]$ is calculated. The completed table for the given sequences is shown below:

			Sequence B								
			D	**C**	**B**	**D**	**A**	**A**	**C**	**B**	**B**
		1	1	1	0	0	0	0	0	0	0
Sequence A	**C**	0	1	1	0	0	0	0	0	0	0
	D	0	0	1	1	1	1	0	0	0	0
	D	0	0	0	1	0	1	0	0	0	0
	C	0	0	0	0	0	1	1	1	0	0
	D	0	0	0	0	0	0	0	1	1	0
	A	0	0	0	0	0	0	0	0	1	0
	C	0	0	0	0	0	0	0	0	1	1

In order for a sequence to be a merged sequence of the given root sequences, the calculated value in cell $T[m, n]$ must be TRUE. In the example above, $T[m, n]$ evaluates to true, so the given sequence is a merged sequence of the root sequences A and B.

Because this algorithm operates iteratively over m elements of sequence A, n elements of sequence B, and then the remaining $(m - 1)$ x $(n - 1)$ elements in Table T, the overall time complexity of the algorithm is $O(mn)$. Furthermore, because the memoized shuffle states are saved in an m x n matrix, the space complexity of this solution is also $O(mn)$.

2. [***Shortest Path Count***]

a) *Describe a dynamic program that determines the distance matrix and counts the number of distinct shortest paths between each pair of vertices of a graph that runs in time O(|V||E|) given the adjacency list for the graph.*

By utilizing the principles of Dynamic Programming and utilizing a memoization table, referred to as a distance matrix, containing lengths and counts of the shortest paths between each pair of vertices in the graph can be calculated in O(|*V*| |*E*|). In order to do this, a series of modified breadth first searches must be performed.

Before beginning the breadth first searches, the data structure for the distance matrix must be created. The matrix must be of size |*V*| x |*V*| where each cell in the matrix contains a pair of values representing the the length of the shortest path and the count of distinct shortest paths of that length. At this point, the space complexity of the algorithm is equal to O(|*V*|^2).

After creating the matrix, a bit of setup must be performed. First, for each cell (*i*, *i*) in the matrix where *i* represents a given node, a value must be calculated. In other words, we must first consider the shortest path from each node in the graph to itself. From an abstract perspective, each node is connected to itself with a path of length 0. Because no other path in the graph can match this length, the count for this path is set to one. Therefore, for each (*i*, *i*) cell in the matrix, a value of (0,1) is inserted. This process executes in |V| time.

Next, a pass through all nodes and edges must be performed in order to obtain all paths of length 1. A single pass is made through the adjacency list and for each pair, the corresponding cells in the table are given a value of (1,1) indicating that there is a path from each of the nodes to the other with a length of 1. The count of the shortest path is assigned a value of one as no other path in the graph can link these nodes with a shorter path length assuming edges between nodes are not duplicated. Because each node in the graph is adjacency list is visited once and because each edge is visited once, the overall complexity of this portion of the algorithm is bounded by O(|*V*| + |*E*|).

At this point, the distance matrix contains shortest path values for all nodes in the graph to themselves and for each node to their immediately adjacent nodes. So far, the algorithm is still bounded by O(|*V*| + |*E*|).

Before progressing, a new data structure must be allocated. This structure will essentially operate as a queue for all adjacent paths and their connected paths for a given node. In actuality, this structure would only contain the node itself, resulting in a space complexity of O(|*V*|), however, to simplify this explanation, we will instead add pairs to the queue where each pair consists of the parent node and the child node.

At this point, the algorithm begins to execute a series of breadth first searches through the adjacency list from each node to every other node. Beginning with the first node, *i*, in the adjacency list, the algorithm begins iterating through each of its adjacent paths. For each node it reaches, the paths from that node to other nodes in the graph are added to the queue. For example, while examining the adjacency list for 1, which may consist of paths to nodes 2 and 3, the paths from node 2 to nodes 3, 4, and 5 will be added to the queue as pairs (e.g., (2,3), (2,4), (2,5)). For each of these pairs a calculation is made to determine the shortest path from *i* to the destination node of the pair. This calculation is performed by summing the path length from *i* to the current source node with the path length from the current source node to the destination node. The resulting path data is then considered with respect to the existing shortest path values in the distance matrix and one of four outcomes are possible:

Case 1: The length of the existing shortest path is less than that of new path
In this case, the newly identified path is ignored because it is a longer path than one previously identified.

Case 2: The length of the new path is less than that of the existing shortest path
Given this case, the shortest path in the distance matrix is replaced with the newly identified path data.

Case 3: The existing shortest path and the new path have equal lengths
Finally, if the two paths are of equal length, the length of the path in the distance matrix is not updated, however, the count of the shortest path is incremented by one. This case essentially identifies another unique path which has a length equal to that of the shortest path.

Case 4: The existing shortest path is null
If the current shortest path is null, the first identified path between the two nodes is used as the shortest path.

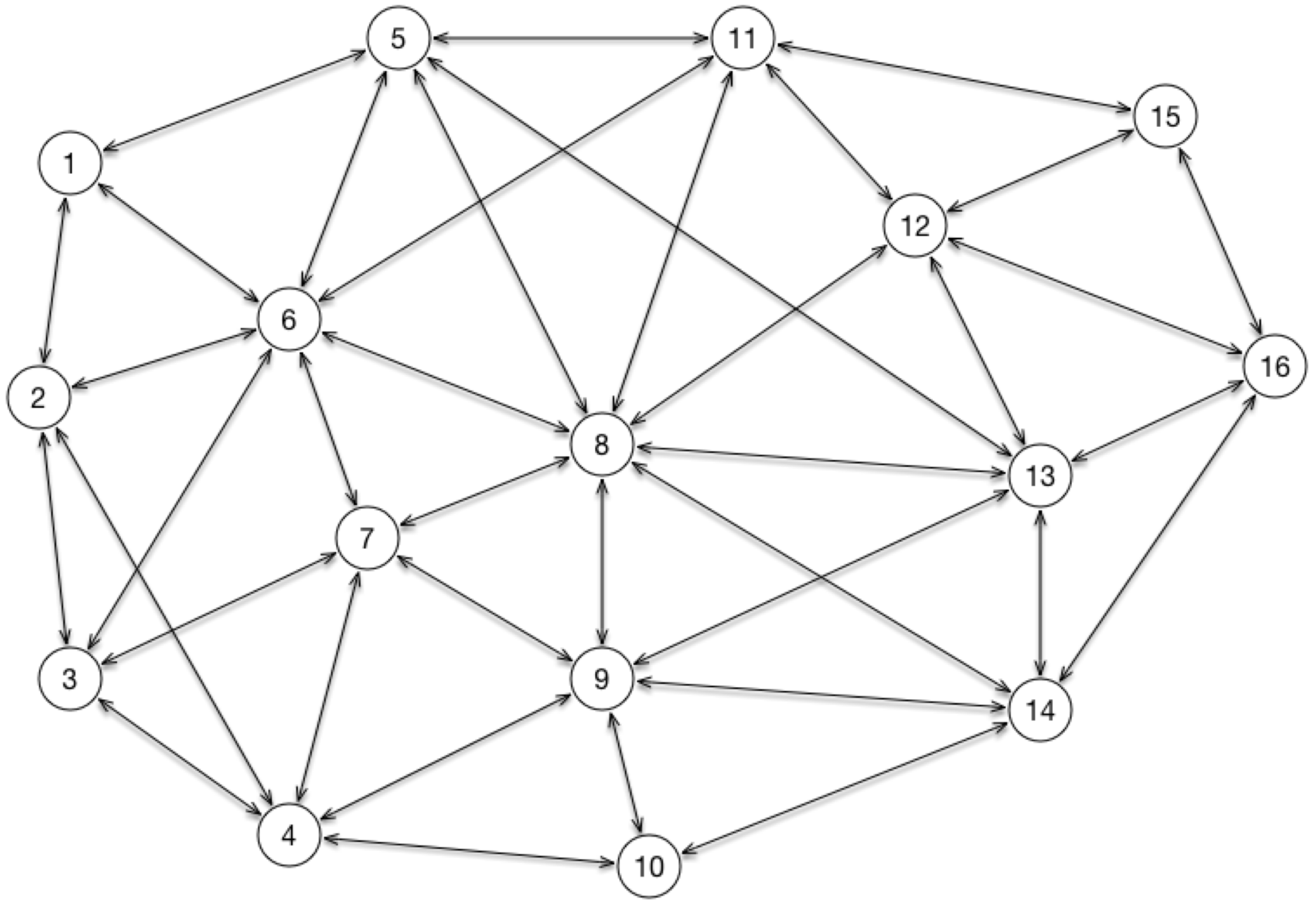
If case 1 occurs, a path longer than a previously identified path is found, at which point the path is discarded and the BFS continues. If either Case 2, 3, or 4 is encountered, however, the distance matrix is updated and the queue must be updated. When this happens, all paths from the destination node to its adjacent nodes are added to the queue. This process continues and the distance matrix continues to be updated until the queue is emptied and all adjacent nodes to *i* have been visited, completing a single BFS.

At this point, the algorithm continues to execute by performing subsequent BFSs for every remaining node in the adjacency list. This process continues until all nodes in the adjacency list has been visited, at which point, the path data in the distance matrix represents the shortest path data for each node to every other node in the graph. This data contains the length of the shortest path between the nodes and it also contains a count for the number of unique paths that exist with that given length.

Each individual BFS described above is bound by $O(|E|)$ and it is performed for each of the $|V|$ nodes in the graph. As a result, the overall time complexity for the algorithm is bound by $O(|V|\,|E|)$. Furthermore, because the largest component of the algorithm's required space is needed for the distance matrix itself, the space complexity for this algorithm is bound by $O(|V|^2)$.

While these complexity bounds are final for the algorithm, there are some optimizations that can be made to reduce the actual space needed for execution. Because the distance matrix consists of path data for each node pair twice (e.g., from A to B and from B to A), the matrix can be split diagonally and the cells below the diagonal can be ignored. As a result, whenever a path is being examined, the "From" node must be the numerically smaller of the pair. The result of this optimization is that only $|V|^2 \div 2$ space is needed for execution, however, as previously stated, the bound remains $O(|V|^2)$.

b) Apply your algorithm to the 16 vertex graph illustrated below.



For purposes of applying the algorithm to the graph illustrated above, first the adjacency list must be created:

1: 2, 5, 6
2: 1, 3, 4, 6
3: 2, 4, 6, 7
4: 2, 3, 7, 9, 10
5: 1, 6, 8, 11, 13
6: 1, 2, 3, 5, 7, 8, 11
7: 3, 4, 6, 8, 9
8: 5, 6, 7, 9, 11, 12, 13, 14
9: 4, 7, 8, 10, 13, 14
10: 4, 9, 14
11: 5, 6, 8, 12, 15
12: 8, 11, 13, 15, 16
13: 5, 8, 9, 12, 14, 16
14: 8, 9, 10, 13, 16
15: 11, 12, 16
16: 12, 13, 14, 15

Next, as stated in the section above, the distance matrix of size |V|^2 must be created; for this example, the resulting distance matrix is size 16 x 16. After creating the matrix, the values for each node's shortest path to itself is entered into the matrix, which is shown in the table below:

		FROM															
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
TO	**1**	0,1															
	2		0,1														
	3			0,1													
	4				0,1												
	5					0,1											
	6						0,1										
	7							0,1									
	8								0,1								
	9									0,1							
	10										0,1						
	11											0,1					
	12												0,1				
	13													0,1			
	14														0,1		
	15															0,1	
	16																0,1

This step of the algorithm requires only a single pass through all nodes, resulting in a complexity which is bound by O(|V|).

At this point, a pass is made through the adjacency list and the path from each node to its immediately adjacent nodes is added to the distance table. This values are represented by a "1,1", which indicates that each of these paths has a length of 1 and is only found once in the graph. Applying this component of the algorithm to the given graph results in the following table:

		FROM															
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
TO	**1**	0,1	1,1			1,1	1,1										
	2	1,1	0,1	1,1	1,1		1,1										
	3		1,1	0,1	1,1		1,1	1,1									
	4		1,1	1,1	0,1			1,1		1,1	1,1						
	5	1,1				0,1	1,1		1,1			1,1		1,1			
	6	1,1	1,1	1,1		1,1	0,1	1,1	1,1			1,1					
	7			1,1	1,1		1,1	0,1	1,1	1,1							
	8					1,1	1,1	1,1	0,1	1,1		1,1	1,1	1,1	1,1		
	9				1,1			1,1	1,1	0,1	1,1			1,1	1,1		
	10				1,1					1,1	0,1				1,1		
	11					1,1	1,1		1,1			0,1	1,1			1,1	
	12								1,1			1,1	0,1	1,1		1,1	1,1
	13					1,1			1,1	1,1			1,1	0,1	1,1		1,1
	14								1,1	1,1	1,1			1,1	0,1		1,1
	15											1,1	1,1			0,1	1,1
	16												1,1	1,1	1,1	1,1	0,1

In order to enter values for the shortest path between each node and its immediately adjacent nodes, the algorithm performs a single pass through the entire adjacency list, which operates in O(|E| + |V|).

Now the BFS component of the algorithm can begin. The algorithm begins with node 1 in the adjacency list. For each immediately adjacent node to node 1, a value pair is pushed into the queue. This process results in the following queue:

(2, 1), (2, 3), (2,4), (2,6) … (5,1), (5,6), (5,8), (5,11), (5,13) … (6, 1), (6, 2), (6, 3), (6, 5), (6, 7), (6, 8), (6,11)

Next, we can remove any pairs that connect to the current node (1), which results in the following updated queue:

(2, 3), (2,4), (2,6) … (5,6), (5,8), (5,11), (5,13) … (6, 2), (6, 3), (6, 5), (6, 7), (6, 8), (6,11)

Now the algorithm pops the first value from the queue, which is (2,3). Because 1 connects to 2 through a (1,1) path and because 2 connects to 3 through a (1, 1) path, the combined path from 1 to 3 is therefore a (2, 1) path. Because this path is shorter than the current path linking 1 to 3 (which is empty), this becomes the new shortest path between 1 and 3. This is shown in the table below:

		FROM															
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
	1	0,1	1,1			1,1	1,1										
	2	1,1	0,1	1,1	1,1		1,1										
	3	**2,1**	1,1	0,1	1,1		1,1	1,1									
	4		1,1	1,1	0,1			1,1		1,1	1,1						
	5	1,1				0,1	1,1		1,1			1,1		1,1			
	6	1,1	1,1	1,1		1,1	0,1	1,1	1,1			1,1					
	7			1,1	1,1		1,1	0,1	1,1	1,1							
TO	**8**					1,1	1,1	1,1	0,1	1,1		1,1	1,1	1,1	1,1		
	9				1,1			1,1	1,1	0,1	1,1			1,1	1,1		
	10				1,1					1,1	0,1				1,1		
	11					1,1	1,1		1,1			0,1	1,1			1,1	
	12								1,1			1,1	0,1	1,1		1,1	1,1
	13					1,1			1,1	1,1			1,1	0,1	1,1		1,1
	14								1,1	1,1	1,1			1,1	0,1		1,1
	15											1,1	1,1			0,1	1,1
	16												1,1	1,1	1,1	1,1	0,1

Because this path from 2 to 3 resulted in an update to 1's shortest paths, the immediate adjacent paths belonging to 3 are pushed into the queue, resulting in the updated queue below:

(2,4), (2,6) … (5,6), (5,8), (5,11), (5,13) … (6, 2), (6, 3), (6, 5), (6, 7), (6, 8), (6,11) … (3, 2), (3, 4), (3, 6), (3,7)

Next, (2,4) is removed from the queue. Because the shortest path from 2 to 4 is (1,1) and because the path from 1 to 2 is (1,1), the path from 1 to 4 through 2 is (2,1). Because this value is less than the current shortest path between 1 and 4 (empty), this path is added to the table and 4's adjacent connections are added to the queue.

		FROM															
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
	1	0,1	1,1			1,1	1,1										
	2	1,1	0,1	1,1	1,1		1,1										
	3	2,1	1,1	0,1	1,1		1,1	1,1									
	4	**2,1**	1,1	1,1	0,1			1,1		1,1	1,1						
	5	1,1				0,1	1,1		1,1			1,1		1,1			
	6	1,1	1,1	1,1		1,1	0,1	1,1	1,1			1,1					
	7			1,1	1,1		1,1	0,1	1,1	1,1							
TO	**8**					1,1	1,1	1,1	0,1	1,1		1,1	1,1	1,1	1,1		
	9				1,1			1,1	1,1	0,1	1,1			1,1	1,1		
	10				1,1					1,1	0,1				1,1		
	11					1,1	1,1		1,1			0,1	1,1			1,1	
	12								1,1			1,1	0,1	1,1		1,1	1,1
	13					1,1			1,1	1,1			1,1	0,1	1,1		1,1
	14								1,1	1,1	1,1			1,1	0,1		1,1
	15											1,1	1,1			0,1	1,1
	16												1,1	1,1	1,1	1,1	0,1

Next, after examining the connection from 2 to 6, a path from 1 to 6 exists as (2,1). Contrary to previous examples, this path is longer than the existing shortest path from 1 to 6, so the connection is not added to the table and 6's adjacent connections are not added to the queue.

If however, the path from 1 to 6 through 2 was of equal length to the current path from 1 to 6, the table would be updated by incrementing the number of shortest paths between the nodes. Similar to if the path was shorter than the current shortest path, the adjacent connections of that node would then be added to the queue.

This process continues until the queue is empty and then proceeds to the next node in the adjacency list. Finally, this process continues until all nodes in the list have been visited and the queue for the last node has been emptied.

c) *Discuss how you can use the results of (a) to find an edge of the graph that has the highest count of distinct shortest paths including that edge.*

In order to calculate which edge belonging to a given graph has the highest count of distinct shortest paths that use that edge, the algorithm must be modified to utilize additional data structures. First, a hash table of |*E*| length must be created. This hash table will essentially store the number of times a given edge was used in a distinct shortest path.

Next, the algorithm must modify the data stored for a given cell in the distance matrix. Because the first shortest path identified which connects two nodes may not necessarily be *the* shortest path or even *one of* the shortest paths, some method of decrementing the usage count for each edge must be created. In order to handle this, whenever the first path between two nodes is identified and added to the table, that cell must add a reference to the edge(s) used in the shortest path(s). At the same time, the value in the hash table at the location of the edge will be incremented by one.

If a an additional shortest path of the same length is identified, that edge is also added to the array of referenced edges and that edge's count in the hash table is incremented. If, however, if a shorter path is identified, before the distance matrix cell can be updated with the new information, the algorithm must iterate through the cell's referenced edges and decrement each edge's value in the hash table. After this has completed, the algorithm continues to perform as described above.

Once the algorithm has finished the entire process, the element of the hash table which has the highest count is the element of the graph which has the highest count of distinct shortest paths including that edge.

d) *(5 point bonus) Find an edge as described in part (c) for the 16 vertex graph of part (b).*

After exhaustively performing the steps described in the section above, the edge with the highest distinct shortest paths including that edge could be found, however, the process required to perform this calculation would be extremely costly in terms of time complexity.

After visually analyzing the graph and taking degree density and position into consideration, it is very plausible that the edge shared between 6 and 8 has the highest count of distinct shortest paths.

3. [***Longest Palindrome Subsequence***]
A palindrome is a nonempty string over some alphabet that reads the same forward and backward, e.g. racecar. Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input CHARACTER your algorithm should return CARAC. What is the running time of your algorithm?

How much space do you need to simply determine the length of a longest palindrome subsequence? [Hint: see text discussion of longest common subsequence.]

Illustrate your algorithm on the following sequence forwards and backwards, to check your solution.
CBAD FEDA EFCB EDEF DEBA DCCA.

Before discussing the method of determining the Longest Palindrome Subsequence (LPS) of a given sequence utilizing the principles of Dynamic Programming, the problem must first be broken down.

In its simplest form, the solution to the LPS problem involves solving a series of sub-problems for a given sequence. These sub-problems, however, continue to generate additional sub-problems until a base case is reached. Unfortunately, without the use of a Dynamic Programming algorithm, this process has an overall efficiency of O(2^n) as a result of generating 2 sub-problems for each problem until the base case is reached.

After giving the problem some thought, a series of patterns begin to emerge regarding the solution of sub-problems. Attempting to find the LPS of a given a string X with length n, which may or may not be part of a sub-problem, can be broken down into a series of cases. Each of these cases is dependent on the value of n and the values of the first and last elements of the string andean be summarized by the following:

$n == 1$

Case 1:
Because the string is only a single letter, the string is a palindrome of length 1.

$P = X$
$m = 1$

$n == 2$

Case 2: $X[1] == X[n]$
In this case, X consists of 2 identical letters and is therefore a palindrome of length 2.

$P = X$
$m = 2$

Case 3: $X[1] \mathrel{!}= X[n]$
X is a string consisting of two unique letters. As a result, the string contains 2 separate palindromes of length 1. Because both palindromes have equal length, P can be assigned to either of the two letters.

$P = X[1]$ OR $X[2]$
$m = 1$

$n > 2$

Case 4: $X[1] == X[n]$
In this case, X is a string consisting of at least 3 letters where the first and last letter of the string are identical. Because the first and last characters are identical, the palindrome must consist of at least 2 characters. Furthermore, because there exists at least one additional character in the string in between the two identical characters, the length of the palindrome must be greater than 2.

In regards to the palindrome itself, the first and last characters of P must be equal to the first and last letters of X. As a result, the following can be said about P:

$P[1] = P[m] = X[1] = X[n]$
$m > 2$

Finally, the remaining elements of P can be found by breaking LPS(X) into an additional sub-problem. Because $X[1]$ and $X[n]$ are components of the LPS, they can be stripped from X, leaving $X[2 \ldots n\text{-}1]$. The solution to the LPS of $X[2 \ldots n\text{-}1]$ will then be used as follows:

P[2 … m-1] = LPS($X[2 \ldots n\text{-}1]$)

$X[1] \mathrel{!=} X[n]$

In this situation, X does not begin and end with the same character. As a result, the LPS of X could potentially include $X[1 \ldots n\text{-}1]$ or $X[2 \ldots n]$.

Case 5: LPS($X[1 \ldots n\text{-}1]$) ≥ LPS($X[2 \ldots n]$)
In the event that $X[1 \ldots n\text{-}1]$ is larger, the following can be gleaned:

P = LPS($X[1 \ldots n\text{-}1]$)
m ≥ LPS($X[1 \ldots n\text{-}1]$)

Case 6: LPS($X[2 \ldots n]$) < LPS($X[1 \ldots n\text{-}1]$)
In the event that $X[2 \ldots n]$ is larger, the following can be gleaned:

P = LPS($X[2 \ldots n]$)
m ≥ LPS($X[2 \ldots n]$)

As previously stated, each sub-problem results in one of the above cases. While these cases may seem very organized and simple to solve, the greater solution involves solving many of these sub-problems, and without Dynamic Programming, each sub-problem is solved multiple times. In order to employ a Dynamic Programming approach, a memoization table will be used to store the solutions to sub-problems, which will be used to eliminate a substantial amount of the overall sub-problems that must be solved.

Using the cases discussed above, the algorithm is applied to the given sequence, which results in the following memoization table:

		C	B	A	D	F	E	D	A	E	F	C	B	E	D	E	F	D	E	B	A	D	C	C	A
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
C	1	1	1	1	1	1	1	3	5	5	5	7	7	7	7	7	9	11	11	13	13	13	15	15	15
B	2	0	1	1	1	1	1	3	5	5	5	5	7	7	7	7	9	11	11	13	13	13	13	13	13
A	3	0	0	1	1	1	1	3	5	5	5	5	5	5	7	7	9	11	11	11	13	13	13	13	13
D	4	0	0	0	1	1	1	3	3	3	5	5	5	5	7	7	9	11	11	11	11	11	11	11	11
F	5	0	0	0	0	1	1	1	1	3	5	5	5	5	5	7	9	9	9	9	9	11	11	11	11
E	6	0	0	0	0	0	1	1	1	3	3	3	3	3	5	7	7	7	9	9	9	11	11	11	11
D	7	0	0	0	0	0	0	1	1	1	1	1	1	3	5	5	5	7	7	7	9	11	11	11	11
A	8	0	0	0	0	0	0	0	1	1	1	1	1	3	3	3	5	5	7	7	9	9	9	9	11
E	9	0	0	0	0	0	0	0	0	1	1	1	1	3	3	3	5	5	7	7	7	7	9	9	9
F	10	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	5	5	5	7	7	7	9	9	9
C	11	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	3	3	5	7	7	7	9	9	9
B	12	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3	3	3	5	7	7	7	7	7	7
E	13	0	0	0	0	0	0	0	0	0	0	0	0	1	1	3	3	3	5	5	5	5	5	5	5
D	14	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3	3	3	3	5	5	5	5
E	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3	3	3	3	3	3	4
F	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	3	3	4
D	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	3	3	4
E	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	2	4
B	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	2	4
A	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	4
D	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	2
C	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	2
C	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
A	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The table above is split diagonally to demonstrate that approximately $n^2 \div 2$ sub-problems are no longer required to calculate the LPS of the sequence.

Using the algorithm and applying it to the given sequence in reverse order results in the following memoization table:

		A	C	C	D	A	B	E	D	F	E	D	E	B	C	F	E	A	D	E	F	D	A	B	C
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
A	1	1	1	2	2	4	4	4	4	4	4	5	5	7	9	9	9	11	11	11	11	11	13	13	15
C	2	0	1	2	2	2	2	2	3	3	3	5	5	7	9	9	9	9	11	11	11	11	13	13	15
C	3	0	0	1	1	1	1	1	3	3	3	5	5	7	9	9	9	9	11	11	11	11	13	13	15
D	4	0	0	0	1	1	1	1	3	3	3	5	5	7	7	7	7	9	11	11	11	11	13	13	13
A	5	0	0	0	0	1	1	1	1	1	3	3	5	7	7	7	7	9	9	9	9	11	13	13	13
B	6	0	0	0	0	0	1	1	1	1	3	3	5	7	7	7	7	7	7	9	9	11	11	13	13
E	7	0	0	0	0	0	0	1	1	1	3	3	5	5	5	5	7	7	7	9	9	11	11	11	11
D	8	0	0	0	0	0	0	0	1	1	1	3	3	3	3	5	5	5	7	7	9	11	11	11	11
F	9	0	0	0	0	0	0	0	0	1	1	1	3	3	3	5	5	5	5	7	9	9	9	9	9
E	10	0	0	0	0	0	0	0	0	0	1	1	3	3	3	3	3	3	5	7	7	7	7	7	7
D	11	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	3	5	5	5	7	7	7	7
E	12	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	3	3	3	5	5	5	7	7
B	13	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	3	5	5	5	7	7
C	14	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	5	5	5	5	7
F	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	5	5	5	5	5
E	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3	3	3	5	5	5
A	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	5	5	5
D	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3	3	3	3
E	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
F	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
D	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
A	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
B	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
C	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Through the utilization of backtracking, the optimal subsequence that forms the Largest Palindrome Sequence can be determined from the memoization tables above. For the given sequence, the LPS, which has a length of 15, is as follows:

[C A D F E D E B E D E F D A C]

While the dynamic programming algorithm used to solve this problem cuts the number of operations and the space needed in half, the overall time complexity and space complexity remain unchanged. Because the algorithm performs $n^2 \div 2$ operations, the overall time complexity of the algorithm is still bound by $O(n^2)$.

Furthermore, in order to simply calculate the length of the longest palindrome, without having to determine what the actual palindrome sequence is, the actual required space is approximately $n^2 \div 2$, which bound by $O(n^2)$.

4. [***Monotonic Sequences***]
Given the sequence a(1), a(2),..., a(n) of real numbers, consider the problem of determining the length k of a longest local maxima subsequence (first increasing monotonically to some element, then decreasing monotonically).

Using the paradigm of dynamic programming give the best algorithm you can for determining the length of a longest local maxima subsequence. Describe the algorithm and analyze its time and space complexity. (See text p. 319, 16.3-5, 6). Discuss the storage space required to solve for just the variable k compared to obtaining a subsequence of that length.

From an abstract perspective, the goal of determining the longest local maxima (LLMA) subsequence for a given sequence might seem trivial, however, in actuality the problem is quite complex. Depending on the distribution of the values in the sequence and their locations in the sequence, the location of the LLMA can vary wildly.

Although the problem is difficult, by utilizing the principles of dynamic programming, the value of length of the LLMA can be found in a very simple manner. For this solution, a memoization table in the form of a dictionary of length *k* where *k* is the length of the LLMA will be utilized.

Before delving into the inner workings of the algorithm, a top-level overview will be detailed. The algorithm essentially utilizes a dictionary of sorted values to keep track of the LLMA at any given point during the traversal of the sequence. Whenever a new element is encountered, it is placed appropriately in the dictionary and when the entire sequence has been processed, the length of the dictionary is equal to the length of the largest monotonically increasing subsequence in the sequence. Next, the algorithm is applied to the sequence in reverse with the length of the resulting dictionary representing the value of the largest monotonically decreasing subsequence in the sequence. The final component of the algorithm involves finding the element of the sequence that, when used as a midpoint between monotonically increasing and monotonically decreasing subsequences, results in the longest LLMA. This final step will be discussed towards the end of this analysis.

The algorithm begins by first creating a dictionary. Any given key in the dictionary will correspond with the respective element in the sequence and will store information about that element. This dictionary will be discussed later in this analysis and will be referred to as the ‘element dictionary’. Next, the algorithm creates a second dictionary, which will be used to hold sorted values of the sequence. When the first item is encountered, it is immediately entered into this dictionary at index 1 as there are no other values in the dictionary to compare it against. Next, the following element is handled by comparing it to the first element in the dictionary. If the current element is greater than the first element in the dictionary, the current element is appended to the dictionary, else, it replaces the first item in the dictionary. This process continues until the dictionary contains 3 values, at which point, all subsequent elements are placed with the aid of a binary search.

The process of placing elements within the dictionary is simple, however, it is a very elegant solution to the problem. As the dictionary is populated, items are inserted in-order and, as a result, whenever a new element is encountered, a binary search can be performed to identify where the new value should be placed. If the current value is greater than the last value in the dictionary, the value is appended. Similarly, if the the current element is smaller than the smallest element in the dictionary, it replaces that element. For all other cases, a binary search is performed to identify the smallest element that is greater than or equal to than the current element. When that element is found, the current element replaces it. Because this step in the algorithm utilizes a binary search, each element encountered results in a time complexity of O(lg*k*), however, because this is performed for all *n* elements in the sequence, the overall time complexity of the algorithm is O(*n*lg*k*).

Regardless of the outcome of inserting an element into the dictionary, after any given element, *j*, is inserted, the current length of the dictionary is calculated. This value represents the length of the longest monotonically increasing from in the subsequence [1 … *j*]. Next, the value in the 'elements dictionary' using a key equal to the current element's value is retrieved and initialized to 0 if it was previously null. At this point, the length of the current longest monotonically increasing subsequence is added to the retrieved value and stored back in the dictionary using the same key. After completion, the process continues with the next element.

After all *n* elements have been encountered and inserted into their appropriate positions in the dictionary, many of which will replace previously inserted elements, the end result is a dictionary of length *k* where the value of *k* is equal to the length of the longest monotonically *increasing* subsequence. As stated above, this process executes with a time complexity of O(*n*lg*k*) and because the algorithm only requires two dictionaries, one of length *n* and another of length *k*, and because $n \geq k$, the overall space complexity is bound by O(*n*).

Next, the entire process will be repeated with the same subsequence, however, it will be applied in reverse. By applying the algorithm to the reversed sequence, the length of the resulting dictionary represents the length of the longest monotonically *decreasing* subsequence. Furthermore, the sorted dictionary will be emptied and re-used for this second pass.

Once both passes through the sequence have completed, the length of the LLMA can be calculated in *n* time by simply finding the max value in the 'elements dictionary'. Because both passes through the sequence added the length of the increasing and decreasing subsequences at that element, the value for each element in the dictionary represent the length of the LLMA where the current element acts as the midpoint between the monotonically increasing and monotonically decreasing subsequences.

Because both passes through the data operate with a time complexity of O(*n*lg*k*), the overall time complexity involved with calculating the length of the LLMA is bound by O(*n*lg*k*). Furthermore, because each pass uses the same dictionary of length *k* and because only one additional dictionary of length *n* is needed, the space complexity of the algorithm is bound by O(*n*).

In order to determine the actual LLMA of length *k* of the sequence, the algorithm needs to modify the way it stores data in the sorted dictionary. Each element of the dictionary must become an object which must maintain information about the element. This information consists of the information discussed above as well as where the element is located in the original sequence. Furthermore, both sorted dictionaries must be preserved; the first cannot be re-used by the second pass through the sequence.

After the algorithm finishes execution, by utilizing the information stored in the dictionary for each element, the actual LLMA can be determined by backtracking. Because each element in the dictionary has info about it's position in the original sequence, the LLMA can be pieced together by only choosing elements that maintain the order of the original sequence. This modification would still result in a space complexity bound of O(*n*).

Apply your algorithm to the following data, forwards and backwards:

54, 76, 30, 44, 74, 15, 78, 67, 36, 46, 11, 77, 42, 49, 82, 73, 80,
66, 52, 58, 22, 68, 35, 40, 24, 13, 55, 27, 39, 16, 43, 93, 61, 53,
94, 49, 74, 45, 60, 83, 18, 73, 42, 69, 67, 22, 61, 30, 63, 51, 62.

As stated in the analysis above, the first step is to create the data structures used by the algorithm.

Element Dictionary = {} // E{}
Sorted Dictionary = {} // S{}

After creating these dictionaries, the first elements are inserted according to the steps of the algorithm described above. A series of initial steps are shown below with the values of the element dictionary and sorted dictionary displayed *after* insertion has completed for both the forwards and backwards application of the algorithm to the given sequence:

FORWARD:

Element: 54
E{54:1}
S{54}

Element: 76
E{54:1, 76:2}
S{54, 76}

Element: 30
E{54:1, 76:2, 30:2}
S{30, 76}

Element: 44
E{54:1, 76:2, 30:2, 44:2}
S{30, 44}

Element: 74
E{54:1, 76:2, 30:2, 44:2, 74:3}
S{30, 44, 74}

Element: 15
E{54:1, 76:2, 30:2, 44:2, 74:3, 15:3}
S{15, 44, 74}

Element: 78
E{54:1, 76:2, 30:2, 44:2, 74:3, 15:3, 78:4}
S{15, 44, 74, 78}

BACKWARD:

Element: 62
E{62:1}
S{62}

Element: 51
E{62:1, 51:1}
S{51}

Element: 63
E{62:1, 51:1, 63:2}
S{51, 63}

Element: 30
E{62:1, 51:1, 63:2, 30:2}
S{30, 63}

Element: 61
E{62:1, 51:1, 63:2, 30:2, 61:2}
S{30, 61}

Element: 22
E{62:1, 51:1, 63:2, 30:2, 61:2, 22:2}
S{22, 61}

Element: 67
E{62:1, 51:1, 63:2, 30:2, 61:2, 22:2, 67:3}
S{22, 61, 67}

The process show above continues to execute until a pass has been made through all elements of the original sequence and the reversed sequence. At this point, the greatest value in the elements dictionary is the length of the LLMA and the key for that value is the point in the sequence at which the monotonically increasing portion of the LLMA ends and the monotonically decreasing remainder of the LLMA begins. For the given data, the value of *k* is 15, indicating the LLMA for this sequence is 15 elements long. Finally, the key associated with this value is 94, therefore the increasing portion of the LLMA ends at 94 and the decreasing portion begins thereafter.

5. [***Randomized Dynamic Subsequence Selection***]

a) *Consider the dynamic ascending subsequence selection problem as illustrated in the SOLO game discussed in class. Write a program to compute the expected value of the 52 card game. Tabulate the solution of the n card game and the "take point" for the 20 values $n = 2^k$, where $k = 1, 2, \ldots, 20$. Estimate functions describing the general solution for the value and take point for arbitrary n.*

The SOLO game is essentially a game in which the player attempts to get the highest number of cards possible where each subsequent card taken must be greater than the card taken before it. Given that the deck is randomly shuffled, depending on the first card chosen, the total number of cards drawn can vary significantly.

In order to maximize the number of cards taken by a player, a “Take Point” is estimated. This take point resembles the point at which a card should or should not be drawn. When a player draws a given card, *i*, the number of remaining cards that player is able to draw is reduced to $n - i$. Because, however, the player added a card to the number of cards they have drawn, the overall representation of the impact of drawing a card can be represented as $1 + (n - i)$. Alternately, the impact of *not* drawing a card can be represented as $(n - 1)$ as the number of remaining cards from which to draw has been reduced by 1.

Using these two representations, an approximation can be made for the estimated value for a deck of cards of size *n*. The function accepts the number of cards in the deck, *n*, and calculates this estimate by breaking the problem into sub-problems. Each sub-problem represents an estimated expected value from the smaller deck. This problem continues until the base case of either 0 or 1 is reached.

Without a dynamic programming approach, the solution to the problem involves calculating and re-calculating subproblems, resulting in an overall time complexity of $O(2^n)$. In order to increase this efficiency, a dynamic programming approach will utilize a memoization table which will store expected values for decks of size n.

In order to determine the expected value for a deck of size *n*, the algorithm will begin with an *i* value of 2 and continue estimating the expected value of a deck with size *i* and continue until *i* equals *n*. At each subsequent estimation, previous expected values are utilized by referencing values in an array of expected values.

For deck of size *n*, the expected value of each card from 1 to *n* is calculated. Each calculation results in a series of sub-problems, the solutions to which are used to evaluate whether it would be more beneficial to take the current card or to ignore it. The estimation for these values is considered and the larger of the two is used. This process continues until an estimation has been performed for each card in the deck, which is then used to calculate the expected value for the entire deck. Using the representations of “taken” and “ignored” cards discussed above and given this explanation, the estimation value for a deck of size *n* can be represented by following formula:

$$V(n) = \frac{1}{n}\sum_{i=1}^{n} max\{1 + V(n-i), V(n-1)\}$$

Finally, in order to calculate the “take point” for a given deck of size *n*, a separate array must be utilized. While the expected value for a given card in the deck is being calculated, a pointer keeps track of which card values for the deck are “taken” and which are “ignored”. The last value of *i* which results in a taken card is the “take point” for that deck.

Applying this algorithm to calculate the expected value, V(n), and the "take point" for the 20 values $n = 2^k$, where $k = 1, 2, \ldots, 20$ results in the data below:

k	***n***	**V(*n*)**	**Take Point**
1	2	1.5	1
2	4	2.375	2
3	8	3.601190	4
4	16	5.331638	5
5	32	7.758678	8
6	64	11.166875	11
7	128	15.953614	16
8	256	22.686190	22
9	512	32.166908	32
10	1,024	45.532121	45
11	2,048	64.389173	64
12	4,096	91.011810	90
13	8,192	128.615856	128
14	16,384	181.749441	181
15	32,768	256.844728	256
16	65,536	362.99828775	362
17	131,072	513.074690295	512
18	262,144	725.267240673	724
19	524,288	1025.3051964	1024
20	1,048,576	1449.57524983	1448

b) (5 points bonus) How little space is sufficient for computing the value of the n card game?

Using the dynamic programming approach described above, the only space required to determine the expected values of an n card game is n. The algorithm essentially makes a pass from 1 to n and calculates the expected value for each card value along the way by using the sub-problem solutions which are stored in previous indices of the array. Furthermore, an additional array of size n must be used if "take point" calculations are needed. Assuming both are calculated, the overall space utilized is equal to $2n$.

Because only $2n$ extra space is needed for computation of the expected values and the "take points", the overall space complexity is bounded by $\boldsymbol{\Theta(n)}$.

Implementation for 5a:

```
import sys

def V(num_cards):
     v = [0,1]                    #Array used for V(n) estimations
     take_points = [0,1]          #Array used for take point estimations
     take_point = 0

     # Calculate values for V(n) up to num_cards
     for n in range(2, num_cards + 1):
          expected_value = 0

          # For the current n, get the expected value for all cards
          for i in range(1, n + 1):
               take_value = 1 + v[n - i]
               ignore_value = v[n - 1]

               expected_value += max(take_value, ignore_value)


               if take_value > ignore_value:
                    take_point = i

          # Divide the summed expected values by n
          expected_value /= float(n)

          # Add estimated values to their respective arrays
          v.append(expected_value)
          take_points.append(take_point)

     print take_points

     for i in range(1, 13):
          print "k: %d" % i
          print "n: %d" % (2**i)
          print "V(n): %f" % v[2**i]
          print "Take Point: %d\n" % take_points[2**i]


def main():
     if len(sys.argv) < 2:
          print "Requires value for n"
     elif len(sys.argv) > 2:
          print "Too many arguments"
     else:
          V(int(sys.argv[1]))

main()
```