Chris Ayala
December 4, 2013

Algorithms HW 5

Question 1) De-Merging

When considering a solution to the problem, we can look at the string C from back to front. If C is a shuffle of A and B, then the last character of C must be equal to the last character of A or the last character of B. If this is true, then the remaining characters in C must be a shuffle of the remaining characters in A and B.

We can come up with a simple recursive algorithm with that logic:

```
bool isShuffle (A, B, C) {
    if (length(A) equals ∅)
        return B equals C
    if(length (B) equals ∅)
        return A equals C

    if(A[n-1] equals C[p-1])
        if ( isShuffle (A[1...n-1], B, C[1...p-1]))
            return true;
    if(B[m-1] equals C[p-1])
        if(isShuffle (A, B[1...m-1], C[1...p-1]))
            return true;
    return false;
```

Lines 1-4 are the base cases, where we have exhausted all characters from one of our two input strings. In those cases, the remaining characters in C must equal the remaining characters in the non-empty string.

At each call of the function, there exists the possibility of 2 recursive calls. In each call, the problem size decreases by 1. Thus, by the master theorem, we have exponential growth.

Rather than branching off as it does, we can save the progress of shuffle states in an $(n+1)\times(m+1)$ table T. Each element $T[i,j]$ is either true or false. It is true if $C[1\ldots(i+j)]$ is a shuffle of $A[1\ldots i]$ and $B[1\ldots j]$. If it is not, then $T[i,j]$ is false. Working backwards, we see that for $T[i,j]$ to be true, one of the following conditions must be true as well:

1) $A[i]$ equals $C[i+j]$, and $C[1\ldots(i+j-1)]$ is a shuffle of $A[1\ldots i-1]$ and $B[1\ldots j]$. As in, $T[i-1,j]$ is true.
2) $B[j]$ equals $C[i+j]$, and $C[1\ldots(i+j)-1]$ is a shuffle of $A[1\ldots i]$ and $B[1\ldots j-1]$. As in, $T[i,j-1]$ is true.

If one of these conditions is true, then $C[1\ldots i+j]$ is a shuffle of $A[1\ldots i]$ and $B[1\ldots j]$, and thus $T[i,j]$ is true. Otherwise, it is false.

Using these conditions, we can formulate a Dynamic Programming algorithm:

```
bool isShuffle(A, B, C)
1     Initialize T as (n+1)x(m+1) bool array
2     Set T[0,0] = true
3     for i = 1 to length(A)
4           T[i,0] = (A[1...i] equals C[1...i]
5
6     for j = 1 to length(B)
7           T[0,j] = (B[1...j] equals C[1...j])
8
9     for i = 1 to length(A)
10          for j = 1 to length(B)
11                T[i,j] = ((A[i] equals C[i+j] AND T[i-1,j]
12                      OR ((B[j] equals C[i+j] AND T[i,j-1])
13
14    return T[n,m]   // using 0-based indices
```

Lines 3-4 traverse string A, which is $\Theta(n)$.
Lines 6-7 traverse string B, which is $\Theta(m)$.
Lines 9-12 iterates over and assigns each element in the nxm matrix in T, and is thus $\Theta(mn)$.

In this case, the table is first set up with the upper- and left-most row and column, respectively. Then, it checks the two previously mentioned conditions on a row-by row basis.

ABBAADCCDBABACD

String B

		B	A	C	D	A	B	D
	1	0	0	0	0	0	0	0
A	1	1	0	0	0	0	0	0
B	1	1	1	0	0	0	0	0
A	0	1	1	0	0	0	0	0
D	0	0	1	1	0	0	0	0
C	0	0	1	1	1	0	0	0
B	0	0	0	0	1	1	1	0
A	0	0	0	0	1	0	1	0
C	0	0	0	0	0	0	1	1

String A

Above is the table T for the sample input. As mentioned previously, we first set $T[0,0]$ to true. Then, we fill in the top row and left column. Finally, in a row-by-row basis, we fill in the table depending on whether lines 11-12 are satisfied for the particular cell. What we see is that $T[n][m]$ is true, thus we have correctly determined that C is a shuffle of A and B. The time and space complexities are both $\Theta(nm)$.

Question 2) Shortest Path Count

Part a)

To apply DP to this problem, we need to maintain a table of distances D, that is of size $n \times n$. In this case, $n = |V|$. We will continuously update this table as we go along with the algorithm.

We will let element $d_{ij}^{(k)}$ be the length of the shortest path between vertices v_i and v_j, having at most k intermediary connections. It may be the case that the actual number of connections between v_i and v_j is less than k. $d_{ij}^{(0)}$ is set to be the the weight matrix entry w_{ij}, since there are no intermediate points between i and j ($k=0$). If these two vertices are not connected, then $d_{ij}^{(0)} = w_{ij} = \infty$. We know that, at most, v_i and v_j will be at most $n-1$ edges apart. Thus, to compute the all-pairs shortest path, we need to compute $D^{(n)}$.

For a given pair of vertices v_i and v_j, looking at entry $d_{ij}^{(k)}$, there are two possibilities:

1) V_k is not a vertex on the shortest path.
   Thus the shortest path $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
2) V_k is a vertex on the shortest path.
   Thus the shortest path $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

Rule 2 applies because if k is an intermediate point between v_i and v_j, then the path must include the subpaths $v_i \rightarrow v_k$ and $v_k \rightarrow v_j$, both of which are optimal shortest paths.

Combining the two rules leads to the following recurrence:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

We can set up an iterative approach with the following algorithm:

```
1  Initialize D to the same values as W
2  for K = 1 to n
3      for i = 1 to n
4          for j = 1 to n
5              if (D[i][j] > D[i][k] + D[k][j])
6                  D[i][j] = D[i][k] + D[k][j]
7  return D
```

We can condense the for loops on lines 3 and 4. Since we have an adjacency list structure for W, as opposed to an adjacency matrix, we rewrite the loops to:

```
3      for each vertex vi in V
4          for each edge ej in vi
```

The remainder of the logic remains the same. What is important about this change is that, with the original algorithm, it was guaranteed to run in $\Theta(|V|^3)$ time. However, lines 3-4 now iterate over edges in the adjacency list. Thus, lines 3-4 operate in $\Theta(|E|)$ time, and thus the entire algorithm operates in $\Theta(|V||E|$ time.

Part b)

Refer to next page.

Here we initialize the table to the adjacency list contents.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a	0	1	∞	∞	∞	∞	∞	∞	1	1	1	∞	∞	∞	∞	∞
b	1	0	1	∞	∞	∞	∞	∞	∞	1	1	∞	∞	∞	∞	∞
c	∞	1	0	1	∞	∞	∞	∞	∞	∞	1	1	∞	∞	∞	∞
d	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	1	∞	1	∞

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a	0	1	2	∞	∞	∞	∞	2	1	1	1	2	∞	∞	2	2
b	1	0	1	2	∞	∞	∞	∞	2	1	1	2	∞	∞	2	2
c	2	1	0	1	2	∞	∞	∞	2	2	1	1	2	∞	2	2
d	∞	2	1	0	1	2	2	2	∞	∞	2	2	1	2	1	2

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a	0	1	2	3	3	∞	3	2	1	1	1	2	3	3	2	2
b	1	0	1	2	3	∞	3	3	2	1	1	2	3	3	2	2
c	2	1	0	1	2	3	3	3	2	2	1	1	2	3	2	2
d	3	2	1	0	1	2	2	2	3	3	2	2	1	2	1	2

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a	0	1	2	3	3	4	3	2	1	1	1	2	3	3	2	2
b	1	0	1	2	3	4	3	3	2	1	1	2	3	3	2	2
c	2	1	0	1	2	3	3	3	2	2	1	1	2	3	2	2
d	3	2	1	0	1	2	2	2	3	3	2	2	1	2	1	2

All sequential tables are identical. Since no weights were provided in the input data set, I assume that all edges have weight 1.

Question 3) Longest Palindromic Subsequence

This question resembles the longest common subsequence problem. In fact, one can solve this problem directly by finding the longest common substring between the input string and its reverse. However, while this can be done in $O(n^2)$, this typically does not utilize dynamic programming.

Instead, we can maintain a table for the optimal subproblems on substrings of the input string X. We will denote a substring as X[i....j], wherein the substring contains all characters between i and j, inclusive. $1 <= i <= n$, $1 <= j <= n$.

Suppose we have an input string X with n characters. We can set up a set of recurrence rules with a string P that is of length $m <= n$, and such that P is some longest palindromic subsequence of X.

1) If $n = 1$, then $P = X$, and $m = 1$

This is, of course, the trivial case of a one character word. It is implied that P is a palindrome of length 1.

2) if $n=2$ and $X[1] == X[2]$, then m is 2 and $P=X$

In this case we have a string of length 2, which is 2 of the same letters. Once again, P is the string X, wherein the entire string is the LPS.

3) If $n=2$ and $X[1] != X[2]$, then m is 1 and $P=X[1]$ or $P=X[2]$.

We have a string of 2 different characters. Thus, our LPS length is 1, and P is either of the two characters.

4) If $n>2$ and $X[1]==X[n]$, then $m>2$, $P[1]=P[m]=X[1]=X[n]$ and $P[2...m-1]$ is an LPS of $X[2...n-1]$

We will look in-depth at this statement. The first implication is that $m>2$. This makes sense because there are $n>2$ characters in X. Thus, our palendrome string P can contain the first and last characters in X (which are stated to be matching), and at least one additional character in X. Thus, $m>2$. The second implication of $P[1]=P[m]=X[1]=X[n]$ is simply stating that, since the last two characters in the subsequence X math, we can add the two characters onto P.

The final implication of $P[2...m-1]$ being an LPS of $X[2...n-1]$ is valid because of the previous implication. We have already shown that the outer two elements of P are the same as the outer two elements of X. Thus, we can break apart the problem into one subproblem, ignoring the outer two characters in P and X. Since P is an LPS substring in X, $P[2...m-1]$ is also an LPS in $X[2...n-1]$, because it is a similar string.

5) If $n>2$ and $X[1] != X[n]$ and $P[1] != X[1]$, then P is an LPS of $X[2...n]$

I repeat that we assume P is an LPS of X. If the trailing letters of X are not equal, then P must be an LPS of a substring in X. In the event that $P[1] != X[1]$, then we know $X[1]$ is not in P, and thus P is in $X[2...n]$. The final statement is symmetrical to 5:

6) If $n>2$ and $X[1] != X[n]$ and $P[1] != X[n]$, then P is an LPS of $X[1...(n-1)]$

With these 6 statements, we can set up a table containing lengths of the LPS in a subsequence $X[i..j]$. The table is upper triangular, of size $i \times j$. It is upper triangular because we don't need to concern ourselves with the case of $j > i$. Each element in the table is filled using the following relation:

$$L[i,j] = \begin{cases} 1 & \text{if } i = j \\ 2 & \text{if } j = i+1 \text{ and } X[i] == X[j] \\ 1 & \text{if } j = i+1 \text{ and } X[i] != X[j] \\ L[i+1, j-1] + 2 & \text{if } j > i+1 \text{ and } X[i] == X[j] \\ \max(L[i, j-1], L[i+1, j]) & \text{if } j > i+1 \text{ and } X[i] != X[j] \end{cases}$$

Line 1 represents situation 1, line 2 → situation 2, line 3 → situation 3, line 4 → situation 4, and line 5 is a combination of situations 5 and 6.

The table L above simply stores the lengths of LPS of each subsequence. Thus, finding the maximal length LPS takes $O(n^2)$ time and space (which will be demonstrated). In order to find what the LPS actually is, we need to construct an additional table of directions. These directions will determine the string itself, which will also become clear.

We can compute the length and directions tables using the following algorithm:

```
LPS (X)
    n = length (X)
    D = n x n table
    L = n x n table

    for i = 1 to n-1
        L[i,i] = 1              // Trivial length 1 case
        if X[i] == X[i+1]       // Length 2, both characters
            L[i,i+1] = 2        // are the same
            D[i,i+1] = ↙
        else
            L[i,i+1] = 1        // Length 2, characters are
            D[i,i+1] = ↓        // different

    for i = n-2 to 1
        for j = i+2 to n
            if X[i] == X[j]     // Situation 4
                L[i,j] = L[i+1,j-1] + 2
                D[i,j] = ↙
            else if L[i+1,j] ≥ L[i,j-1]   // Line 5 of
                L[i,j] = L[i+1,j]         // previous page
                D[i,j] = ↓                // (Situation 6)
            else
                L[i,j] = L[i,j-1]
                D[i,j] = ←

    return D and L
```

The directions are used to maintain where in the table L[i,j] is coming from. The case of the ↙ direction is the optimal direction: in this case, we can extend our LPS.

Once we have the tables D and L, we can easily construct the LPS by starting at element D[1, n], and following the arrows. When we see a ↙ direction, we can surround the current LPS with that character.

```
LPS(D, X, i, j)
1     if (i > j) return empty string
2     if (i == j) return X[i]
3     if (D[i,j] == ↙)
4         return X[i] + LPS(D, X, i+1, j-1) + X[i]
5     if (D[i,j] == ↓)
6         return LPS(D, X, i+1, j)
7     if (D[i,j] == ←)
8         return LPS(D, X, i, j-1)
```

As previously mentioned, we only append characters in the case of ↙. Otherwise, we simply move down a row (i+1), or left a column (j-1). The initial call would look like: LPS(D, X, 1, n).

With that algorithm in hand, we can use it to construct our own L and D tables for the input string "BFADFBECDAEBFCACEFDBAEDC".

First, we look at the L-table (in a following page). What we see immediately is that the longest LPS is of length 13. As mentioned, this is an upper-triangular matrix. The diagonal is computed first (which is assumed to be all 1's trivially), followed by the super-diagonal (which is also all ones because no two characters are sequentially repeated).

Next, we look at the p-table (also in a following page). Starting at the right-most and top-most element (P[1, n]), I have outlined the path by following the arrows. At every 3, a.k.a. ↙, we prepend and append the current LPS with X[i], which is outlined below the table. We stop once we reach a Ø. What we see is that the LPS for this string is "CDEBFCACFBEDC", which is indeed 13 characters long.

In terms of complexity, it is easy to see that we require 2 nxn matrices of space; thus, it is $\Theta(n^2)$ space complexity. Similarly, computing $\frac{1}{2}(n \times n)$ elements takes $\Theta(n^2)$ time complexity. If we are ONLY computing the length of the LPS, then we don't need the D table, and thus require half the space (though it's still $\Theta(n^2)$)

L-Table

1	1	1	1	3	5	5	5	5	5	5	7	7	7	9	9	9	11	11	13	13	13	13	13
0	1	1	1	3	3	3	3	3	5	5	5	7	7	9	9	9	11	11	11	13	13	13	13
0	0	1	1	1	1	1	1	3	5	5	5	7	7	9	9	9	9	11	11	13	13	13	13
0	0	0	1	1	1	1	1	3	3	3	5	7	7	7	7	7	9	11	11	11	11	13	13
0	0	0	0	1	1	1	1	1	1	3	5	7	7	7	7	7	9	9	9	9	11	11	13
0	0	0	0	0	1	1	1	1	1	3	5	5	5	5	5	7	7	7	9	9	11	11	13
0	0	0	0	0	0	1	1	1	1	3	3	3	3	3	5	7	7	7	7	9	11	11	13
0	0	0	0	0	0	0	1	1	1	1	1	1	3	3	5	5	5	7	7	9	9	11	13
0	0	0	0	0	0	0	0	1	1	1	1	1	1	3	3	5	5	7	7	9	9	11	11
0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	3	5	5	5	7	9	9	9	9
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	5	5	5	7	7	9	9	9
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	3	5	5	7	7	7	7	7
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3	3	5	5	5	5	5	5	5
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	3	3	3	3	3	3	3	3	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	3	3	3	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	3	3	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

D-Table

0	1	1	1	1	3	2	2	2	1	1	3	1	1	1	1	1	1	3	1	1	1	1
0	0	1	1	3	2	2	2	1	1	1	1	3	1	1	1	1	3	1	1	1	1	1
0	0	0	1	1	1	1	1	1	3	2	1	1	1	3	2	2	1	1	1	3	2	1
0	0	0	0	1	1	1	1	3	2	1	1	1	1	1	1	1	1	3	2	2	1	3
0	0	0	0	0	1	1	1	1	1	1	1	3	2	2	2	1	3	2	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	3	2	2	2	1	1	1	1	3	1	1	1
0	0	0	0	0	0	0	1	1	1	3	2	2	1	1	1	3	2	1	1	1	3	1
0	0	0	0	0	0	0	0	1	1	1	1	1	3	1	3	1	1	1	1	1	1	3
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	3	1	1	1	2
0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	1	1	1	1	1	3	1	1
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	3	1	1	1	1	3	2
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	3	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	3	2	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	2	2	2	2	1	1	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	3	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

B F A D F B E C D A E B F C A C E F D B A E D C

1 = ↓

2 = ←

3 = ↙

CDEBFCACFBEDC

Question 4) Monotonic Subsequences

I will first discuss how to find the length k of the longest monotonically decreasing subsequence (LMDS) of a sequence. Then I will expand the algorithm to include finding the actual sequence itself.

For a given sequence $a_1, a_2, \ldots, a_n$ the strategy is to find the maximum LMDS length at each a_i. In particular, we want to find what the LMDS length is for the first i elements, ending with a_i. We can set up the following rules:

1) If $i = 1$, then the subsequence is of length 1, and the LMDS is one character long.

2) Otherwise, the LMDS at element i is the maximum LMDS for all elements $j < i$ plus 1, such that $a_j \geq a_i$. If there are no elements $a_j \geq a_i$, then the LMDS at element i (ending in a_i) is 1.

This can be more clearly expressed as a recurrence, where we maintain a length array L:

$$L[i] = 1 + \underbrace{\max_{j<i \;\&\; a_j > a_i} L[j]}_{\emptyset \text{ if no such element } a_j > a_i \text{ exists}}$$

Using the two rules and the recurrence, we can set up an iterative solution to fill the length array L and find the length k of the LMDS.

```
1  Initialize L with n elements
2  L[1] = 1
3  for i = 2 to n
4       L[i] = 1
5       for j = 1 to i-1
6            if (a[j] > a[i]) AND ((1 + L[j]) > L[i])
7                 L[i] = 1 + L[j]
8
9  max = 0
10 for i = 1 to n
11      if max < L[i]
12           max = L[i]
13 return max                    // this is our length k
```

We can clearly see that lines 3-7 is a $\Theta(n^2)$ operation, since at every index i we check all elements before element i. Lines 10-12 is $\Theta(n)$, and thus our algorithm is time complexity $\Theta(n^2)$. In terms of space, we require an n element array L, and is thus $\Theta(n)$.

We can actually easily determine the actual LMDS itself by maintaining a second table of positions P. Essentially, when we update a value L[i] on line 7, we store the index j in P[i]. This index represents the previous element that is the LMDS ending element not including a[i]. We will use this structure to backtrack when we find the LMDS itself. The algorithm rewritten:

```
Initialize L and P with n elements
L[1] = 1
P[1] = 0
for i = 2 to n
    L[i] = 1
    P[i] = 0
    for j = 1 to i-1
        if (a[j] > a[i]) AND ((1 + L[j]) > L[i])
            L[i] = 1 + L[j]
            P[i] = j

return L and P.
```

Once the function returns, L contains the lengths of the LMDS ending with element a_i, and P contains position chains for these LMDS.

With these two sets, we can go through the following algorithm to find the LMDS:

```
1  index = index of largest element in L
2  while index != ∅
3        prepend a[index] to LMDS
4        index = P[index]
5  return LMDS
```

We prepend on line 3 because we are traversing the LMDS in reverse order.

What we see in terms of complexity is that we require twice as much space as before, since we added on the table P (though it is still $\Theta(n)$). Time complexity remains unchanged at $\Theta(n^2)$, because the traversal above is bounded by $\Theta(n)$.

Applying the algorithm forward to the sequence below:

Sequence: 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

L: 1, 1, 2, 2, 2, 3, 1, 3, 4, 4, 5, 2, 5, 4, 1, 3, 2, 4, 5, 5, 6, 4, 6, 6, 7, 8, 6, 7, 7, 8, 7, 1, 5, 7, 1, 8, 3, 9, 6, 2, 10, 4, 10, 5, 6, (11), 7, 11, 7, 8, 8

P: 0, (0), 1, 1, (2), 3, 0, (5), 8, 8, 9, 7, 10, 8, 0, 5, 15, 8, 18, 18, (13), 16, 13, 13, 23, 25, 20, (23), 24, 25, 27, 0, 18, (27), 0, (34), 12, (36), 33, 32, 38, 37, (38), 42, 44, (43), 45, 43, 45, 34, 49

We can repeat the algorithm backwards for the same fundamental results:

Sequence: 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

L: 6, (11), 3, 5, 10, 2, 11, 9, 5, 4, 1, 10, 4, 4, 10, 9, 9, 8, 5, 7, 2, 7, 3, 3, 2, 1, 6, 2, 2, 1, 3, 7, 6, 5, 7, 4, 6, 3, 3, 6, 1, 5, 2, 4, 3, 1, 2, 1, 2, 1, 1

P: 34, (5), 28, 13, (16), 26, 12, 18, 23, 13, 0, 16, 24, 38, 17, (18), 18, (20), 36, (27), 41, 33, 28, 29, 46, 0, (34), 46, 48, 0, 43, 40, 34, (36), 40, (38), 42, (43), 43, 42, 0, 44, (48), 45, 49, 0, 50, (0), 51, 0, 0

Thus, in both cases, our LMDS is:

76, 74, 73, 66, 58, 55, 53, 49, 45, 42, 30

which is of length 11

Question 5)

As discussed in class, when a card with a particular value i appears, we need to decide if we should accept or reject that card. If we choose that card, then in all subsequent cards we can only choose $(n-i)$ possible cards, and plus 1 for having chosen that card. If we choose not to take that card, then we still have $n-1$ cards to choose from. Thus the resulting score is the same as if we had $n-1$ cards.

I will define $V(n)$ as the expected score for n cards. At a given card i, we can compare $V(n-i)+1$ (this is to take the card) and $V(n-1)$ (this is to ignore the card. We will choose the larger of the two, so that we maximize our score. For a set of n cards, each card has a probability $\frac{1}{n}$ of being chosen. Over all cards i, our expected score $V(n)$ can be expressed with the following recurrence:

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

Our base cases for this recurrence are the following two values:

$$V(0) = 0$$
$$V(1) = 1$$

Using that recurrence relation, I have written a short program that will calculate these values. As an example, for $n = 52$ is:

$$V(52) = 10.021$$

The solutions to the n card game, where $n = 2^K$ having K in [1, 18], is shown below:

K	n	V(n)	Take point
1	2	1.5	2
2	4	2.375	2
3	8	3.60119	4
4	16	5.33164	5
5	32	7.75868	8
6	64	11.1669	11
7	128	15.9536	16
8	256	22.6862	22
9	512	32.1669	32
10	1024	45.5321	45
11	2048	64.3892	64
12	4096	91.0118	90
13	8192	128.616	128
14	16384	181.749	181
15	32768	256.845	256
16	65536	362.998	362
17	131072	513.075	512
18	262144	725.267	724