Problem Set 7

Hongrui(Ray) Liu

Exercise:

6.1. In this particular problem, we are required to find the maximum sum of a contiguous sequence in a list of n numbers.

Firtst, we need to have two indexes: cur_head and cur_tail to traverse the current contiguous sequence. If the sum of the current sequence is bigger than the sum we are supposed to return(indexed by sum_head and sum_tail), we update both sum_head and sum_tail to the same values as cur_head and cur_tail.

And also, we should think about when we should give up the current sequence and jump to the next sequence. Let's say we are given a list: 5, 15, -30, 10, -5, 40, 10.

And we can notice that the sum of the current sequence can also be the largest even if the occurance of negative numbers. It should be clear that only if the sum with the negative number is not less than 0, the current sum can still proceed to get bigger and remain the largest. For example. 5, 15 is in the current sequence. After -30 is added to the sum, the sum drops below 0. And no matter how large the following numbers are, its sum would not be larger than the sum that starts at the next number. Therefore, if the current sum is less than 0, we should move the current starting index to the next one.

Here's the pseudocode for this algorithm:

Input: a list S of n integers Output: a subsequence of S which has the maximum sum

max_sum = MIN_INFINITY // store the largest sum

cur_sum = 0 //store the sum of the current subsequence

cur_head = 0 // the starting index of current subsequence

cur_tail = 0 // the ending index of current subsequence

sum_head = 0 // the starting index of maximum sum

sum_tail = 0 // the tailing index of maximum sum

for i = 0 to n - 1

```
cur_tail = i
cur_sum += S[i]
if cur_sum > max_sum
    max_sum = cur_sum
    sum_head = cur_head
    sum_tail = cur_tail
```

```
  //else : sum is less than 0
  if cur_sum < 0
      cur_sum = 0
      cur_head = i + 1
```

return S[sum_head, sum_tail]

Time Complexity: O(n) Therefore, the algorithm given above is a linear algorithm.

6.5

(a) According to the statement made in the question, we shouldn't put pebbles vertiacally adjacent in a column. Therefore, there are actually 7 patterns in a isolation:

pattern 0
number
number
number
number

pattern 1
-Pebble-
number
number
number

pattern 2
number
-Pebble-
number
number

pattern 3
number
number
-Pebble-
number

pattern 4
number

pattern 4
number
number
-Pebble-

pattern 5
-Pebble-
number
number
-Pebble-

pattern 6
-Pebble-
number
-Pebble-
number

pattern 7
number
-Pebble-
number
-Pebble-

(b)

In order to better illustrate the key point, I drew a chart listed below as an example:

And there are 3 columns in this particular example

Checkerboard Layout

C1	C2	C3
1	50	70
15	23	45
60	90	26
10	125	100

In this case, the optimal replacement matrix would be :

	C1	C2	C3

	C1	C2	C3
P0			
P1			
P2			
P3			
P4			
P5			
P6			

Different sums for column1 would be :

	C1	C2	C3
P0	1	61	
P1	15		
P2	60		
P3	10		
P4	11		
P5	61		
P6	25		
P7	12		

As for the second column, we need to take adjacency into consideration. Take Pattern0 as an example. Since no pebble is placed onto the checkerboard, it can match with any pattern in the column one. As a result, the sum at (P0, C2) would be the largest sum(61) in the C1 plus the value at (P0, C2) which is 0. So (P0, C2) would be uddated as 61. Similarly, we can do this at (C2, P1) but notice that it doesn't match with sum at (P1, C1), (P5, C1), (P6, C1).

Based on the example given above, we can derive the following algorithm for this problem:

Input:

Matrix[Pattern][Column] for the checkerboard where Pattern = 8, it can show the pebbled value at the position (Pn, Cn)

Initialize an array Match[i] to determine wether the current pattern can match with the previous pattern.

And Sum[Pattern][Column] repsents the sum at the Sum[i][j]

Output : the sum for the given checkerboard

Pseudocode:

For column 1 to N :

```
For Pattern 0 to 7
    TMP = 0 // temporary max value
    a1 = Matrix[column][Pattern]

    For column - 1(the previous column) in Match[Pattern] // for all the
patterns in the previous column that match the current pattern

        a2 = Sum[column - 1][Pattern]

        if a1 + a2 > TMP
            TMP = a1 + a2

    Sum[Pattern, column] = TMP
```

//When we jump out of the nested loop, it means we have completely traverse the whole checkerborad, so we just need to return the maximum value in the last column.

return MAX(Sum[i][N]) while i ranges from 0 to 7

Time Complexity: Although there are 3 loops in the given algorithm, the runtime for 2 inner loops should be O(8 * 8) which can be considered as a constant. Therefore, the time complexity for this algorithm would be O (N * constant) = O(N)

6.8

Let's say we are given two strings : string s1 = "longest and string s2 = "stone". And if we stop at 'o' for "longest" and 'o' for "stone", the LCS length would be 1. And we proceed to the next character. We stop at 'n' for both strings and at this point, we the LCS length at 'n' should be 1 + LCS length at the previous position.

We can use a matrix to better illustrate this idea.

			l	**o**	**n**	**g**	**e**	**s**	**t**
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
s	1	0	0	0	0	0	0	1	1
t	2	0	0	0	0	0	0	1	2
o	3	0	0	1	1	1	1	1	2
n	4	0	0	1	2	2	2	2	2
e	5	0	0	1	2	2	3	3	3

And the sub-problem to find the LCS length is :

if (A[i] == B[i]) LCS[i][j] = LCS[i - 1][j - 1]

else LCS[i][j] = MAX(LCS[i - 1][j], LCS[i][j - 1])

Here's the pseudocode for this algorithm:

Input : a matrix[i][j] where the values at matrix[0][j] and matrix[i][0] are all initialized as 0

			l	**o**	**n**	**g**	**e**	**s**	**t**
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
s	1	0							
t	2	0							
o	3	0							
n	4	0							
e	5	0							

Output : the length of the LCS

for i = 1 to m

```
for j = 1 to n

    if (A[i] == B[i ])
      LCS[i][j] = LCS[i - 1][j - 1]

    else
      LCS[i][j] = MAX(LCS[i - 1][j], LCS[i][j - 1])
```

return matrix[m][n]

Time complexity: Since there are two nested loops, the time complexity would be O(m * n)

6.18

(i,j)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	1	0	0	0	0	0	0
2	1	1	1	1	0	0	0	0
3	1	1	1	2	1	1	0	0

The matrix given about represents the number of ways to make change for v.

First, we initialize the first column to be 1, since we can choose not to take any coins to form a total value 0.

As for the first row, the a(i,j) would be valued 1 if a[i] equals a[j]. For example, a[1, 1] = 1.

And then for each position in the matrix, we should first determine whether the value of i is larger than the j. If so, we just need to copy the value from previous column(j - 1). Otherwise, the value at the current position = the value from previous row(i - 1) + Find_Num(the total value(v) - the current coin value(i)) where Find_Num(i) represents the number of ways to make change for i.

But we should notice that one type of coin can not be used twice, when we run Find_Num(the total value - the current coin value), we should determine whether (the total value - the current coin value) is not less than i. If it is, we should return 0, otherwise, we should return the corresponding value.

Input : Matrix[i][j] represents the number of ways to make change for v and i <= m j <= n

Output : true or false (true means the given denominations can make change for v while false means the other way around)

//First, we should initialize the fist row, set Matrix[i][i] = 1 for the first row : Matrix[i][i] = 1

//Then initialize the first column. for the first column : Matrix[i][0] = 1

for i = 2 to m

```
for j = 1 to n

    if j - i < i // no repetition
        Matrix[i][j] = Matrix[i - 1][j] + Matrix[i][j - i]
    else

        Matrix[i][j] = Matrix[i - 1][j]
```

if Matrix[m][n] > 0

```
return true
```

return false