**1. Describe in English how you can break down the larger problem into one or more smaller problem(s). This description should include how the solution to the larger problem is constructed from the sub problems.**

The key here is to be able to identify the current state of the partitioned original array.

a. We know since we need to keep each partition to be equal or less than the input value ( t ) we need a way to keep track of the sum in which we used a variable.

b. In order to store the results of an inequality score of some particular sub array we need to have a 2D array to store those values.

c. We need to be able to check all of the possibilities of every combination of allowable partitions (not a doomed state) and be able to compare all of the different combinations of inequality scores and return the lowest sum of those values using the min function.

**2. What recurrence can you use to model the problem using dynamic programming?**

Our function MemoEP with a parameter datasetC which is being modified recursively

{MemoEP(datasetC[i+1..n], t, Sindex+1, n) if, sum <= t && ep[Sindex][1..n] == inf

MemoEP(datasetC, t,Sindex,n) = { min(ep[Sindex][1..n]) if, sum <= t && ep[Sindex][1..n] != inf

{ep[Sindex][Nindex] = inf - 1 if, sum > t

{0 if, Sindex > n

**3. What are the base cases of this recurrence?**

Once we leave our bounds for the original array datasetC ( start index > n) return 0

**4. Describe a pseudocode algorithm that uses memoization to compute the inequality score (i.e., the sum of the squares of the unused capacity) for the optimal (least unequal) solution.**

**Input**: *dataset*: array of numbers  
**Input**: *t*: max partition size  
**Output**: minimum inequality score for an array of numbers



**Input**: *datasetC* : array being passed, will change per recursive call  
**Input**: *startingIndex:* used to keep track of what row of ep is being manipulated  
**Input**: *n*: size of original array  
**Output**: minimum value in the *startingIndex* row of the array *ep*



**5. Describe an iterative algorithm for the same purpose**

Input: *dataset* : array of integers  
Input: *t* : max partition size  
Output: minimum inequality score for the array



**6. Describe how to modify and extend your iterative algorithm to identify the optimal partition.**

Input: *dataset* : array of integers  
Input: *t* : max partition size

Input: n: size of dataset  
Output: number of partitions followed by each partition size

1 **Algorithm** IterEP

2 *n* **=** size of dataset **-** 1

3 EP **=** Array**(***n***,***n***)**

4 Initialize EP to MAX\_INT

5 **for** i **=** 0 to n

6 **|**

7 **|** sum **=** 0

8 **|** **for** j **=** i to *n*

9 **|** **|** sum **=** sum **+** dataset**[**j**]**

10 **|** **|** **if** sum **<=** t

11 **|** **|** **|**

12 **|** **|** **|** **if** i **!=** 0

13 **|** **|** **|** **|** EP**[**i**][**j**]** **=** (**(**t**-**sum**)**\*(t-sum)) **+** min**(**EP**[**0...*n***][**i**-**1**])**

14 **|** **|** **|** **else**

15 **|** **|** **|** **|** EP**[**i**][**j**]** **=** **(**t**-**sum**)\*(t-sum)**

16 **|** **|** **|** end

17 **|** **|** **else**

18 **|** **|** **|** **break**

19 **|** **|** end

20 **|** end

21 end

22

23 Initialize stack

24 trueMin = MAX\_INT

25 **for** i=0 to n do

26 **|** **if** trueMin > EP[i][n]

27 **|** **|** trueMin = EP[i][n]

28 **|** **end**

29 **end**

30 column = n, row = 0, and stackCount = 0

31 **while** column greater or equal to 0

32 **|** stackMin = MAX\_INT

33 **|** **for** i=0 to n

34 **|** **|** **if** stackMin > EP[i][column]

35 **|** **|** **|** stackMin = EP[i][column]

36 **|** **|** **|** row = i

37 **|** **|** **end**

38 **|** **end**

39 **|** push (column – (row + 1)) onto stack

40 **|** column = row – 1

41 **|** increment stackCount

42 **end**

43 output stackCount

44 **while** stack not empty

45 | output top of stack then pop

46 **end**

**7. Analyze the time and space complexity of your improved iterative algorithm.**

In reference to the above pseudocode from above (problem 6)

Lines that are not mentioned are negligible (i.e. end).

Lines that are constant: 2, 7, 9, 10, 12, 13, 15, 23, 24, 25, 26, 27, 30, 32, 34, 35, 36, 39, 40, 41, 43, 45

First loop: lines 5-21: inner loop lines 8-20: The body of the loop is constant time and iterates O(n) times. Therefore the inner loop has a complexity of O(n). The outer loop lines 5-21, has a body complexity of O(1) + O(n) which simplifies to O(n). There are n iterations therefore, the complexity of the entire for loop lines 5-21 is O(n2).

The next loop (lines 25-29) has a body complexity of constant. It iterates n times therefore, the complexity of the loop is O(n).

The next loop (lines 31 – 42) is nested. The inner loop, lines 33-38 has a constant body run time and iterates n times for a complexity of O(n). The outer loop body is then constant + O(n) = O(n). The main loop lines 31-42 iterates n times with a body complexity of O(n) yielding a O(n2) complexity.

The final loop lines 44-46 has a constant body complexity and it iterates n times yielding a complexity of O(n)

Outside of the loops the only significant complexity is from lines 3 and 4 which each have a O(n2) complexity. Adding all of these complexities together yields: O(2\*n2 + n2 + n + n2 + n) which simplifies to a total complexity for the entire algorithm to O(n2)

**8. Can the space complexity be improved relative to the memoized algorithm? Justify your answer.**

Yes. We can implement two 1D arrays of length n. One array, called *scores[],* holds the minimum inequality scores of any combination of subsets from the beginning of the original dataset to the nth element of the dataset. The other 1D array called *size[],*stores the partition size corresponding to a subset of the original dataset containing the kth element with minimum possible inequality score.

Our memorized algorithm uses a 2D array which is n x n and has a space complexity of O(n2).

The improvement would have a complexity of O(2n) = O(n)