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 subproblems.

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.
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] -= 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.

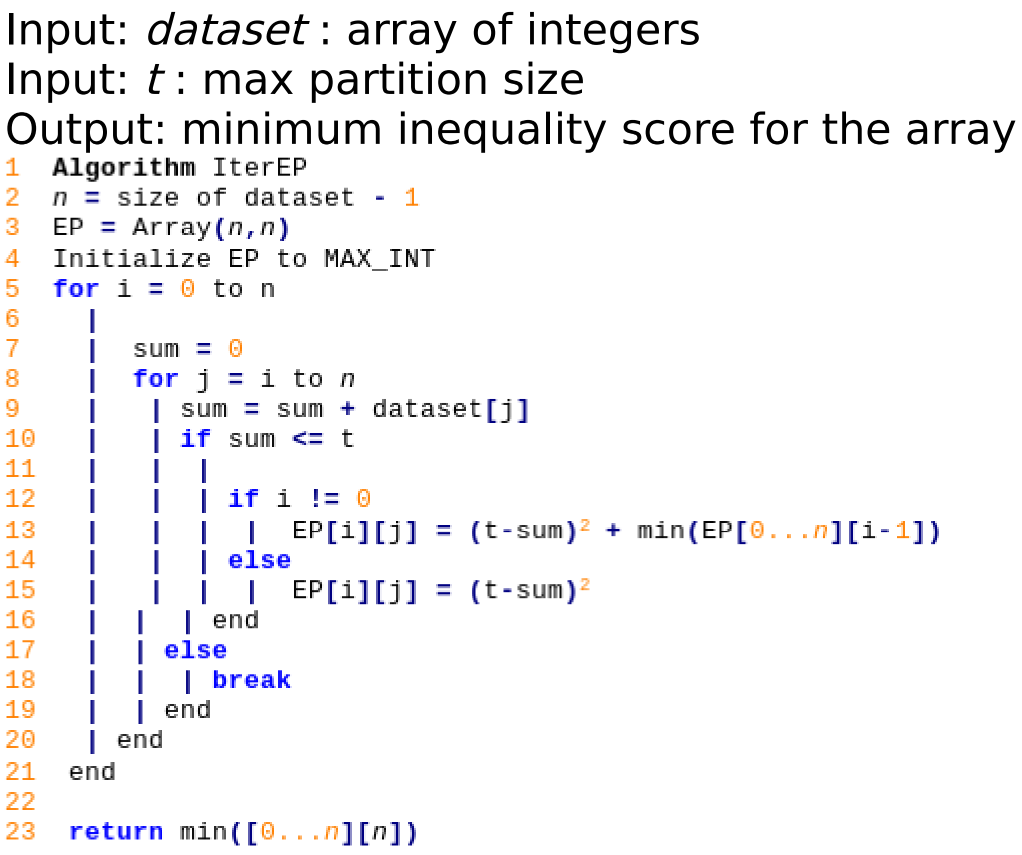
Input: *dataset*: array of numbers
Input: *t*: max partition size
Output: minimum inequality score for an array of numbers

```
Algorithm: EP
ep = Array(n,n)
Initialize ep to infinity
return MemoEP(dataset, t, 0, n)
```

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*

```
Algorithm: MemoEP
int nextIndex = startingIndex;
int sum = 0;
if startIndex <= n
  | for i to the end of the someArray
  |  |   sum += datasetC[i]
  |  |   if(sum <= t)
  |  |    |  if ep[startingIndex][0...n] all != infinity
  |  |    |   |
  |  |    |   |   return min ep[startingIndex][0....n]
  |  |    |   |
  |  |    |  else
  |  |    |   |
  |  |    |   |   ep[startingIndex][nextIndex] = (t- sum(datasetC[startingIndex...nextIndex])
  |  |    |   |   + MemoEP(datasetC[i+1...n], t, nextIndex + 1, n)
  |  |    |   |
  |  |    |  end
  |  |   else
  |  |    | ep[startingIndex][nextIndex] -= 1
  |  |   end
  |  |   nextIndex++
  | end
else
 |   return 0;
end
return min ep[startingIndex][startingIndex...n];
```

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

```
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] = $(t-sum)^2$ + min(EP[0...n][i-1])
14    |    |   | else
15    |    |   |   |  EP[i][j] = $(t-sum)^2$
16    |   |   | end
17    |   | else
18    |   |   | break
19    |   | end
20    | end
21   end
22
23   return min([0...n][n])
```

6. Describe how to modify and extend your iterative algorithm to identify the optimal partition.
**** We already have the optimal partition as described above. Ask Hendrix what he wants here. ****

7. Analyze the time and space complexity of your improved iterative algorithm.
Looking at the code above from question 5
<u>Lines 2, 7, 9, 10, 12, 13, 15, 18, and 22 take constant time</u>

<u>The inner loop body</u> : All of the lines in the body of this loop take constant time<u>O(log(n))</u>.

<u>The inner for loop</u> iterates j – I times which worst case is n times (during the first iteration) yielding a worst case of O(n) but on average yielding n/2 iterations. The total complexity of the inner for loop with the iterations is <u>O(nlg(n))</u>

<u>The outer for loop</u> iterates n times. With a body time complexity of O(nlg(n) + 1) = O(nlg(n))
The total complexity of lines 5-21 is n(nlgn) = n^2lgn = O(n^2lgn)

Adding the rest of the code outside of the loops lines 3 and 4 take n^2 time and everything else is constant. N^2 + n^2lgn = O(n^2lg(n))

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

answer.