Project 2

“Dynamic Programming”

Maximiliano Maurente
Austin Wasilewski
Joshua Baker

1) Before solving this version of the partition problem, the subproblems needed to be identified. To break down the larger problem, we considered every possible partition starting at each individual array element. For each hypothetical partition, we stored the inequality score in a 2D array called "`DParray`". Next, we considered the minimum inequality score for the data elements already computed. For example, if we are on the 3rd data element, then we considered the best partitioning for elements 1 and 2. We do that until the last of the data elements, where we will have already partitioned the array optimally.

2) `dpArr[i][j] = dpArr[row][i - 1] + ((t - sum)  * (t - sum));`
This recursion uses the value of the previous column and current row in `dpArr`, along with the minimum inequality score to compute the value for the current row and column stored in `dpArr`.

3) The base cases of the recurrences are when each of the data elements in the array are partitioned alone, giving a maximum inequality score.

4)

```
T - max(given)
Row -
Col -
minSum - minSum so far as you go through array
data[n] - given array of size n
DParray[n][n] - dynamic programming array

Partition(T, data[n], Row, Col, minSum)
      int sum = o
      int col2 = Col

      //FIRST, FILL IN CURRENT ROW

      while(sum <= t && col2 < n)
            sum += data[col2]
            if(sum <= 10)
                  DParray[Row][col2] = minSum + (t - sum)^2
                  col2++

      //NEXT, FIND THE MIN ELEMENT IN THE COLUMN
        WHERE COLUMN = ROW ... WE PASS IT INTO THE
        NEXT CALL OF THE FUNCTION

      h = heap()
      for(int r = 0, r <= Row, r++)
            h.add(data[r][Col]

      int min = h.min() //MIN OF THE COLUMN THAT THE NEXT CALL WILL RELY
```

ON.

```
      if(Col = n)
            return min
      Partition(T, data[n], Row, Col, min)
```

5) To solve the problem with an iterative algorithm, we use for loops to iterate through and populate the 2-D array, "`DParray`" that will hold all the possible inequality scores. As we move through the elements in the data array to populate `DParray`, we take the minimum of the inequality scores for the elements that have already been processed.

6) As we iterate through the array, we record the corresponding dividing positions for the min inequality score. Once we are finished, we find the minimum inequality score in the array and the corresponding dividing positions which were previously stored. With this information, we can construct the optimal partition.

7) Total complexity of Θ(n^2). See source code for individual complexities.

8) The space complexity can be improved slightly for the memoized solution. Our dependencies are only dependent on the current and previous column. We can declare a 2d row size of 2. Our columns continue to get smaller after every iteration but still have a size max equal to n, our initial input for the first iteration.