Problem Analysis

This problem can be solved using dynamic programming. Let w[0], w[1], …, w[N-1] be the weights and p[0], p[1], …, p[N-1] the positions of the heaps and dp[x][k] the cost of grouping the initial heaps from heap 0 to heap x into k new heaps. Clearly all values such that k >= x + 1 are 0, because we can just leave all heaps in their initial locations. In order to get the other values, it's easy to see that there is no point in making some of the final K heaps in a location where originally there was no one. With that in mind the rest of the values of the table can be computed using this recurrence:

$$dp[x][k] = \min_{y} dp[y][k-1] + \sum_{i=y+1}^{x} w[i] * (p[x] - p[i]), y < x$$

This DP statement comes from the fact that any grouping of k heaps of the first x + 1 elements (here x refers to index in a 0-based indexed array) is the result of performing some grouping of k-1, and moving the remaining heaps to the last one (remember we are only allowed to walk to the right).
Getting such formula is not enough the attack the problem, since a naive implementation of the minimization step will take linear time and thus, the whole solution will be **O (N^2 K)** which is too

slow in this case. In order to get an efficient solution we need to go a little bit deeper on the recurrence. We can rewrite it in this way:

$$dp[x][k] = \min_{y} dp[y][k-1] + \sum_{i=y+1}^{x} w[i] * p[x] - \sum_{i=y+1}^{x} w[i] * p[i] \text{ , } y < x$$

$$dp[x][k] = \min_{y} dp[y][k-1] + p[x] \sum_{i=y+1}^{x} w[i] - \sum_{i=y+1}^{x} w[i] * p[i] \text{ , } y < x$$

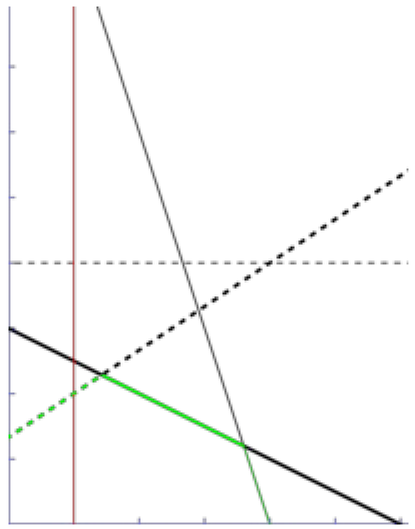
Let $A[i] = \sum_{k=0}^{i} w[k]$ and $B[i] = \sum_{k=0}^{i} w[k]*p[k]$ then, plugging them into the recurrence we have:

$$dp[x][k] = \min_{y} dp[y][k-1] + p[x]*(A[x] - A[y]) - (B[x] - B[y]) \text{ , } y < x$$

and reordering we have:

$$dp[x][k] = p[x]* A[x] - B[x] + \min_{y} p[x]*(-A[y]) + dp[y][k-1] + B[y] \text{ , } y < x$$

With this, we have turned the problem into minimizing a linear form or in other words, for a given value of x (p[x] in this case) we want the lowest available value from a set of lines in the form **y = mx + n**, with m = -A[y] and n = dp[y][k-1] + B[y]. Let's focus now in solving this new problem: suppose that a large set of linear functions in the form y = $m_i x + n_i$ is given along with a large number of queries. Each query consists of a value of *x* and asks for the minimum value of *y* that can be obtained if we select one of the linear functions and evaluate it at *x*. For example, suppose our functions are y = 4, $y = 4/3 + 2x/3$, y = 12 − 3x, and $y = 3 - x/2$ and we receive the query x = 1. We have to identify which of these functions assumes the lowest y-value for x = 1, or what that value is. (It is the function , $y = 4/3 + 2x/3$ assuming a value of 2.)



Graphical representation of the lines in this example

Consider the diagram above. Notice that the line *y* = 4 will *never* be the lowest one, regardless of the *x*-value. Of the remaining three lines, *each one is the minimum in a single contiguous interval* (possibly having plus or minus infinity as one bound). That is, the heavy dotted line is the best line at all *x*-values left of its intersection with the heavy solid line; the heavy solid line is the best line between that intersection and its intersection with the light solid line; and the light solid line is the best line at all *x*-values greater than that. Notice also that, as *x* increases, the slope of the minimal line decreases: 4/3, -1/2, -3. Indeed, it is not difficult to see that this is *always* true.

Thus, if we remove "irrelevant" lines such as $y = 4$ in this example (the lines which will never give the minimum y-coordinate, regardless of the query value) and sort the remaining lines by slope, we obtain a collection of N intervals (where N is the number of lines remaining), in each of which one of the lines is the minimal one. If we can determine the endpoints of these intervals, it becomes a simple matter to use binary search to answer each query.

As we have seen, if the set of relevant lines has already been determined and sorted, it becomes trivial to answer any query in **O (log N)** time *via* binary search. Thus, if we can add lines one at a time to our data structure, recalculating this information quickly with each addition, we have a workable algorithm: start with no lines at all (or one, or two, depending on implementation details) and add lines one by one until all lines have been added and our data structure is complete.

Suppose that we are able to process all of the lines before needing to answer any of the queries. Then, we can sort them in descending order by slope beforehand, and merely add them one by one. When adding a new line, some lines may have to be removed because they are no longer relevant. If we imagine the lines to lie on a stack, in which the most recently added line is at the top, as we add each new line, we consider if the line on the top of the stack is relevant anymore; if it still is, we push our new line and proceed. If it is not, we pop it off and repeat this procedure until either the top line is not worthy of being discarded or there is only one line left (the one on the bottom, which can never be removed).

How, then, can we determine if the line should be popped from the stack? Suppose l_1, l_2, and l_3 are the second line from the top, the line at the top, and the line to be added, respectively. Then, l_2 becomes irrelevant if and only if the intersection point of l_1 and l_3 is to the left of the intersection of l_1 and l_2. (This makes sense because it means that the interval in which l_3 is minimal subsumes that in which l_2 was previously). Let's consider the lines as 2D points with **x = m** and **y = n**, it's not too hard to see that the condition above is equivalent to say that those 3 points are in counter-clockwise order.

Now that we already know how to solve the lines problem, it's pretty much easier to deal with our original task: since we compute the values in the table from left to right, p[x] is strictly increasing, and also, when considering the possible lines, the values of A[y] are increasing too (they are cumulative sums of positive values), hence -A[y] values will be in decreasing order as we process them so we don't need to sort them at all. We get then for the general case, a sketch like this one (see solution for implementation details):

```
for k = 1 to K
   D.clear()
   for x = 0 to n-1
     dp[x][k] = p[x]*A[x] - B[x] + query( p[x] )
     insert((-A[x],  dp[x][k-1] + B[x])) // notice k-1

query(x) goes like this:
while get(x, D[0]) > get(x, D[1])  // get(x, line) is just line.m * x + line.n
  D.pop_front()
return get(x, D[0])

And finally, insert(m, n)
while points D[D.size()-2], D[D.size()-1], (x=m, y=n) are in CCW order
  D.pop_back()
D.push_back( (x=m, y=n) )
```

It's not difficult to see that *query* and *insert* both run in amortized constant time (as every line is inserted just once and discarded just once) which leaves us a complexity of **O(N K)** which is the best possible solution for this problem.