Code Jam 2017 - Round 1C

Analysis: Ample Syrup

Ample Syrup: Analysis

Small dataset

With at most ten pancakes to consider, we can easily enumerate and check all subsets of size **K**, perhaps using a library function such as Python's `itertools.combinations`. When we are considering a subset, the statement's rules tell us exactly how to stack them: in nondecreasing radius order from top to bottom. So all we need is a way to calculate a stack's exposed pancake surface area.

Except for the top pancake, any pancake that is not completely covered by the one above it exposes a ring-shaped outer area of its upper surface. It is possible to calculate and sum the areas of these rings, but there is a much easier method. Observe that the exposed top areas of all pancakes in the stack exactly add up to the top area of the bottom pancake in the stack. That is, if you were to look down on the exact center of the stack, ignoring the heights of the pancakes, the view would be indistinguishable from the top of the bottom pancake. So the exposed surface area of a stack is equal to the top area of the bottom pancake, plus the combined areas of all pancakes' sides. This is $\pi \times \mathbf{R}^2$ for the bottom pancake, plus the sum (over all pancakes in the stack) of $2 \times \pi \times \mathbf{R_i} \times \mathbf{H_i}$. The largest such sum that we find after checking all possible pancake subsets is our answer.

Large dataset

For the Large dataset, we cannot afford to check every possible subset, so we need a better approach. Suppose that we choose a certain pancake P to be on the bottom of our stack. Then every other pancake in the stack must have a radius no larger than the radius of P. Recall from our area-calculating simplification above that the other pancakes besides P effectively only contribute their sides to the total exposed surface area, so we do not need to think about their tops. So, out of the pancakes besides P, we should choose the **K** - 1 of them that have the largest values of $\mathbf{R_i} \times \mathbf{H_i}$.

So, we can try every pancake as a possible bottom; once we choose a possible bottom, the criterion above tells us exactly which other pancakes we should stack on top of it. We can simply search the list for these each time we try a new bottom, given the low maximum value of **N**, but we can do better. For example, we can start with one list of pancakes sorted in nonincreasing order by side area, and another list of pancakes sorted in nonincreasing order by radius. Then we can go through the list of possible radii in decreasing order, treating each pancake in turn as the possible bottom; for each one, we choose the first **K** - 1 pancakes from the list sorted by side area. Whenever we encounter a pancake in the list sorted by side area that comes earlier in the list sorted by radius than our current possible bottom does, we can remove it forever from the list sorted by side area. It is always safe to do this. If that pancake's radius is larger than the radius of our possible bottom, we cannot use it now or ever again, since all future possible bottoms will have a radius that is no larger. If that pancake's radius is equal to the radius of our possible bottom, we have already tried to use that pancake as a bottom previously (since it is earlier in the list sorted by radius), so we have already checked an equivalent stack in which pancakes of the same radius differed only in their order in the stack. Of course, if we encounter our current possible bottom in the list sorted by side area, we should remove that too, because we cannot use the same pancake twice in a stack.

This strategy drops the time complexity to O(**N** log **N** (for the sorts) + **N** × **K**), which is effectively O($\mathbf{N}^2$) in the worst case. We can further improve upon this by storing our best set of **K** - 1 as a min heap / priority queue bases on index in the radius list, so that we do not need to check all **K** - 1 values each time to see whether they have "expired". This drops the complexity to O(**N** log **N** + **K** log **K**), which is equivalent to O(**N** log **N**).