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**).

Code Jam 2017 - Round 1C

Analysis: Parenting Partnering

Parenting Partnering: Analysis

It is clear that whenever one parent has an activity, the other parent must care for the baby throughout that activity. But how should Cameron and Jamie divide up their remaining time after all activities are covered?

Small dataset

In the Small dataset, there are at most two activities, so we can use casework:

- If only one parent has an activity, an optimal solution is to have the other parent care for the baby in a continuous 720 minute block that includes that activity. Then the parent with the activity can handle the other 720 minutes. So only 2 exchanges are required.
- If both parents have one activity each, the same idea as above works. It is always possible to choose some way to divide the 24-hour cycle exactly in half, such that Cameron's activity falls completely within Jamie's half and vice versa. So the answer is again 2.
- If one parent (without loss of generality, we'll say it's Cameron) has two activities, then they divide the 24-hour cycle such that there is a gap between activity 1 and activity 2, and a gap between activity 2 and activity 1. Jamie has to cover both activities. If one of these gaps is empty (which can occur if the activities are back-to-back) or can be completely filled in by Jamie, then the split the day in half strategy works again and the answer is 2. But if the activities are too far apart and/or too long, Jamie may not have enough remaining time to fill in either gap; in that case, both gaps must contain a switch from Jamie to Cameron and back, so the answer is 4. (We will explain later how we can fill in such gaps without adding more exchanges than we need to, regardless of how much time each parent contributes to filling in the gap.)

It does not take much code to implement the above arguments. However, care must be taken to handle midnight correctly when calculating the lengths of the gaps in the third case.

Large dataset

Let's consider a list of the activities, sorted by time. To gracefully handle the midnight transition and the time before the first activity / after the last activity, we will add a copy of the first activity to the end of the list. In this list, each interval is surrounded by two activities. Some intervals may be instantaneous transitions between activities that are back-to-back (let's call these "empty"); the other intervals represent all of the non-activity time. Each of these other intervals is either surrounded by activities covered by different parents, or by activities covered by the same parent.

We have no decisions to make about the empty intervals, but we must count up the exchanges they add. What about the different-parent intervals? For each, we will have to add some time from one or both parents to cover the interval; we can always do this optimally by starting with the time (if any) from the parent covering the activity on the left, and ending with the time (if any) from the parent covering the activity on the right. This strategy always leaves us with just one exchange. We can do no better than that (since the two different parents guaranteed that there would be at least one exchange), and we have no reason to do worse than that. So we do not need to worry about the different-parent intervals at all! We can assume that whatever time we have available (from one or both parents) can fill them up without creating any new exchanges.

Same-parent intervals require some more thought. If we can fill one in with time from the parent who is covering the endpoint activities, we can avoid an exchange, whereas if we include any time at all from the other parent, we will add *two* more exchanges. (We can avoid adding more than two by putting the time from the other parent in in one continuous chunk.) We may not be able to avoid the latter, depending on how much available time each parent has left, but we should fill as many intervals with "matching" time as we can, to minimize the number of new exchanges we create. Each interval contributes equally to the number of possible exchanges, but the intervals may have different lengths, and longer ones take up more of a parent's available time. So our best strategy is to sort the list of Cameron-bounded intervals and greedily fill in the shortest ones with Cameron time until we do not have enough Cameron time left to fill in the shortest remaining interval. Then, we do the same for the Jamie-bounded intervals. For every interval we fail to fill in this way, we add two exchanges to our total.

After we have handled the same-parent intervals, we are done and our current total number of exchanges is our answer. Whatever remaining time we have from one or both parents can safely go into the different-parent intervals without creating any new exchanges. As explained above, we do not need to think about the details; we leave those as an exercise for Cameron and Jamie.

This method is O(**N** log **N**) (because of the required sorting operations) and is easily fast enough for the Large dataset.

Code Jam 2017 - Round 1C

Analysis: Core Training

Core Training: Analysis

Small dataset

In the Small dataset, every core must succeed. The probability that this will happen is the product of the success probabilities of the individual cores: $\mathbf{P_0} \times \mathbf{P_1} \times ... \times \mathbf{P_{N-1}}$. How can we assign our training units to maximize that product?

We will show that it is always best for us to spend our units on the core with the lowest success probability. Suppose that $\mathbf{P_0} < \mathbf{P_1}$, and we have some tiny amount Q of units to spend; we will denote $\mathbf{P_2} \times ... \times \mathbf{P_{N-1}}$ as OtherStuff. If we spend the Q units to improve $\mathbf{P_0}$, our success probability becomes $(\mathbf{P_0} + Q) \times \mathbf{P_1} \times$ OtherStuff. If we instead improve $\mathbf{P_1}$, the product becomes $\mathbf{P_0} \times (\mathbf{P_1} + Q) \times$ OtherStuff. Expanding those expressions, we find that they are identical, except that the first has a $Q \times \mathbf{P_1} \times$ OtherStuff term where the second has a $Q \times \mathbf{P_0} \times$ OtherStuff term. Since we know that $\mathbf{P_0} < \mathbf{P_1}$, the first value must be larger. This means that we should improve $\mathbf{P_0}$ first.

The argument above has one issue to iron out: what if increasing $\mathbf{P_0}$ causes it to rise above $\mathbf{P_1}$? By the same argument, we should switch to improving $\mathbf{P_1}$ instead at the instant that $\mathbf{P_0}$ exceeds $\mathbf{P_1}$, and vice versa if they change ranks again, and so on. So, once we have made $\mathbf{P_0}$ equal to $\mathbf{P_1}$, we should increase both of them at the same time. More generally, we should increase the smallest probability until it exactly matches the next smallest probability, then increase those at the same time until they exactly match the next smallest, and so on. We could try to simulate adding tiny units bit by bit, but it is faster to directly calculate how many units are spent at each step. We must take care to correctly handle the case in which we do not have enough units to fully complete a step.

Large dataset

In the Large dataset, **K** of the cores must succeed. Now it is no longer necessarily optimal to improve the smallest probability. For example, suppose that **N** = 2, **K** = 1, and **U** = 0.01, and the $\mathbf{P_i}$ values for the two cores are 0.99 and 0.01. If we spend all of our 0.01 units on the first core, its success probability becomes 1 and we succeed regardless of whether the second core succeeds. There is no reason to consider spending any units on the second core.

Let's extend this strategy of investing most heavily in a subset of the cores and ignoring the rest. We will sort the cores' success probabilities from lowest to highest, and focus on the ones from some index i onward. As in the Small solution, we will start by improving the success probability of the core at index i to match the success probability of the core at index i+1, then improve those two until they match the success probability of the core at index i+2, and so on. (If all of the success probabilities including and beyond index i become 1, then we can improve the core at index i-1, and so on.) We will show that, for some value of i, this is the optimal strategy. We can then try every possibility for i and keep the one that yields the largest overall answer. Notice that, for one optimal i, at most one "previous" core i-1 needs to be improved, because there is at most one i such that capacity is enough to improve cores i, i+1, ..., **N** up to probability 1 and have some left to improve i-1 but not up to 1 (otherwise, we can just choose i-1 instead).

First, let us consider whether it is better to improve core i or core i+1. Let A_i and B_i be the probability of exactly **K**-2 and **K**-1, respectively, of the cores 1, 2, ..., i - 1, i + 2, i + 3, ..., **N** succeeding. Let $P_{i,d}$ be the probability of at least **K** cores succeding if the success probability of core i is P_i+d and the probability of success of any other core j is P_j. By replacing the definitions and cancelling out some values, we can see that $P_{i,d} - P_{i+1,d} = (A_i - B_i) \times (P_{i+1} - P_i) \times d$. Since $(P_{i+1} - P_i) \times d$ is positive, improving core i+1 is better than improving core i if and only if $B_i > A_i$. Moreover, this doesn't depend on the initial success probabilities of cores i+1 and i, but only on their relative values. So, if we improve core i+1 a little bit, it is still better to keep improving core i+1 if we can instead of switching to improve core i.

Now we want to show that there is some i_0 such that $B_i > A_i$ if and only if $i \geq i_0$. That i_0 is the core where we want to start improving. Notice that A_i depends on **N**-2 probabilities, and A_{i+1} depends on other **N**-2 cores, but **N**-3 of those overlap. Assume fixed **N**-3 core probabilities and let A(p) be the probability that exactly **K**-1 of the **N**-3 fixed cores and an additional core with probability p succed. Define B(p) similarly. We will show that A(p + d) - B(p + d) > A(p) - B(p) for all p and d. Let U, V and W be the probabilities of having exactly **K**-3, **K**-2 and **K**-1 successes out of the fixed **N**-3 cores. Then, A(p) = U × p + V × (1 - p) and B(p) = V × p + W × (1 - p). Then, B(p) - A(p) is a linear function on p, which means if it ever changes from positive to negative, it must be B(0) - A(0) > 0 and B(1) - A(1) < 0, which implies W > V and V < U. However, this is impossible, because it is a well known fact in probability theory that the function f(k) defined as the number of successes in a given set of independent experiments is convex, so it has no local minimum at **K**-2.

With the above claim established, we can check all possible values of i and then take the largest overall success probability that we find. To compute the probability of at least K successes, we can use a dynamic programming method similar to the one described in the analysis for last year's Red Tape Committee problem.

It is difficult to prove this method under time pressure in a contest, but we can discover it via brute force simulation, e.g., by dividing up the available units into quanta and exploring all possible ways to partition them among a small number of cores, and seeing which assignment yields the greatest chance of success. It is also possible to arrive at the correct answers via other methods such as gradient descent.