French Fries Solution

In the French Fries problem, we have (initially P) people repeatedly splitting their fries in half, and handing them out evenly to their neighbors. The process repeats T times, and we ask ourselves after it concludes how many people end up with at least some given number of fries, within a generous error margin.

There are several ways we can approach answering this question, corresponding to the different subtasks.

Naive solution: $O(PT^2)$

For the two simplest subtasks we can simply simulate the process. If coordinates are small we can keep the number of fries for the people at index $[-100, 200]$, and update these in T rounds. If coordinates can be larger things become slightly trickier, but e.g. using a map of some sort should work.

Here is an example solution.

Better solution using precomputation: $O(PT)$

For the next two subtasks, we observe that it is enough to compute how a single fry gets distributed – we can then apply the same distribution pattern to all P fries that get handed out.

In fact, this pattern is pretty simple. The fry gets split T times, resulting in 2^T pieces of size 2^{-T} each, and the number of such pieces that move to the left neighbor k times (and thus to the right neighbor $T-k$ times) is exactly equal to T choose k.

There are, however, a bunch of pitfalls to computing the values $2^{-T}\binom{T}{k}$ for large T. We could use simulation (compute the entirety of Pascal's triangle up to row T), but this takes times $O(T^2)$. We could also use the formula $\binom{T}{k} = T!/k!/(T-k)!$, and pre-compute all factorials between 0! and T!, but this runs up against a fun problem: T! overflows a double for $T > 170$ and becomes infinity.

There are several possible ways one could work around this:

- You could maintain a separate exponent, and increase/decrease it when the number gets too large.
- You could start computing values from the middle, and move from $2^{-T}\binom{T}{k}$ to $2^{-T}\binom{T}{k+1}$ by multiplying by $(T-k)/(k+1)$. We don't know the first value, but we do know that all the values should sum to 1. Thus, we can start by setting middle value to an arbitrary value, and then afterwards divide all values by the resulting sum.
- (Probably the simplest option.) Use logarithms. Then the double exponent never becomes very large, while everything works out the same. This even allows us be lazy and use the method `lgamma` (a builtin in most languages), to compute the logarithm of the factorial function: `lgamma(x) = log((x-1)!)`

Example solution.

Cutting tails: $O(P\sqrt{T})$

If we stop to think about how Pascal's triangle looks, we note something: the values at its left and right ends are rather small. In fact, if T is large, only about the $10\sqrt{T}$ entries nearest the middle are relevant; the rest are almost zero when divided by 2^T. This is a consequence of the Central Limit Theorem, which says that the distribution gets closer and closer to a normal distribution when T gets large. The normal distribution has a standard deviation of $\sqrt{T}$, and has tails that go to zero very rapidly.

The way we can use this is to take the previous solution, and when applying the pattern we only include values close to the middle – this results in a runtime of $O(P\sqrt{T})$.

A full solution

We know how to compute the (approximate) distribution of a single fry, but it is roughly $\sqrt{T}$ entries wide, and we need to apply it in P places... How can we do this quickly? Why, by approximating!

If we separate even and odd indices, the distribution looks very smooth, so we can approximate it by, say, a piece-wise constant function. Having done this, we can proceed by sweeping: at any point where we switch from one piece to another, we add an event. We then sweep through all events from left to right. Between events, we know what constant value the approximated function attains, and we can check whether this value is above or below L. At an event, we can update what the constant value of the function is.

This also extends naturally to larger coordinates and values of T, and precision can be increased by using linear, quadratic, etc. polynomials for the approximation.

Example solution.

Bonus note: Fast Fourier Transforms

It is worth mentioning another method for dealing with this problem: the Fast Fourier Transform (FFT). The basic operation that we perform when splitting a fry into two is a *convolution* (essentially, a linear transform applied at every point of the sequence), and convolutions are very natural to deal with in so-called Fourier space.

If we let C denote the maximum coordinate size, we can use the fast Fourier transform to move all coordinates to Fourier space in $O(C \log C)$ time, perform black magic to do all the convolutions in $O(C \log T)$ time, and then transform back to normal coordinates in $O(C \log C)$ time, for a total runtime of $O(C(\log C + \log T))$.

In practice it turns out to be very hard to get a solution with this time complexity to pass, but for the curious an implementation can be found here.