Code Jam 2018 - Round 1A

Analysis: Waffle Choppers

Test set 1

In test set 1, we are asked to make only one horizontal cut and only one vertical cut. There are **R** - 1 possible horizontal cuts and **C** - 1 possible vertical cuts, so there are a total of (**R** - 1) × (**C** - 1) different ways to make the two cuts. Since **R** and **C** are both at most 10, there are at most 81 ways, and we can try each one of them separately. For each way, we can count up the chocolate chips in each of the four resulting pieces. If we ever find a way in which that number is the same for all four pieces, the answer is `POSSIBLE`; otherwise, it is `IMPOSSIBLE`.

Test set 2

In test set 2, we may have to make many horizontal and vertical cuts, and there are too many ways to do this to check. The worst cases occur when **H** is close to half of (**R** - 1), and **V** is close to half of (**C** - 1); there could be over 10^{57} ways of making the cuts! So we need another approach.

Let us consider only the horizontal cuts, ignoring the vertical cuts for the moment. These horizontal cuts break the waffle into two or more horizontal "slices". Each of these slices will turn into exactly **V** + 1 pieces when we make our vertical cuts. Here is the critical observation: if the case is `POSSIBLE`, all of those pieces must have exactly the same number of chips, and since each horizontal slice will produce exactly the same number of pieces, all of those horizontal slices must have exactly the same number of chips. Moreover, we know exactly what that number is; if there are a total of C chips in the waffle, each of the **H** + 1 horizontal slices must have exactly C / (**H** + 1) chips, or else the case is `IMPOSSIBLE`.

This observation is so powerful that it tells us where we have to make the cuts if the case is `POSSIBLE`! We can make a list of the numbers of chips in each row, then turn that list into a cumulative sum of the total number of chips seen so far. For example, for a waffle with seven rows containing 3, 7, 6, 2, 2, 0, and 10 chips, respectively, the cumulative sum list would be: [3, 10, 16, 18, 20, 20, 30]. In that example, if **H** = 2, we know we have to make cuts immediately below rows that bring the total sum to 10 and 20, and we can do so by cutting immediately below the second and fifth rows. (Note that we could instead make our second cut below the sixth row, but this would make no difference.) If **H** = 1, though, we need to make our one cut immediately below a row that brings the total sum to 15, and there is no such row. So, after this step, either we will know that the case is `IMPOSSIBLE`, or we will know exactly where to make our horizontal cuts.

Then, we can consider only vertical cuts, using a similar method, and either learn where to make them, or learn that the case is `IMPOSSIBLE`. Even if we know where to make the horizontal and vertical cuts, though, we are not done yet! These cuts might not actually create pieces with the same number of chips. For example, for **H** = 1, **V** = 1, and this waffle:

```
..@@
..@@
@@..
@@..
```

we will learn that if the case is `POSSIBLE`, we must make our horizontal cut below the second row, and our vertical cut to the right of the second column. But this creates two pieces with four chips each and two pieces with no chips at all, so the case must be `IMPOSSIBLE`.

To check that each piece has the same number of chips, we can start by turning the list of horizontal cuts into a list of intervals; for example, if we cut below the second and fifth rows in the [3, 10, 16, 18, 20, 20, 30] example from above, then we have the inclusive intervals [1, 2], [3, 5], and [6, 7]. We can do the same for the vertical cuts, and then perform a double iteration over these two sets of intervals, checking each cell within each pair of intervals. We know that each piece must have exactly (total # of chips) / ((**H** + 1) × (**V** + 1)) chips if the case is `POSSIBLE`, so if we ever find a slice in which this is not true, the case is `IMPOSSIBLE`. Otherwise, we have finally shown that the case really is `POSSIBLE`. (Once again, note that even if we had a choice of where to make one or more of our cuts, this must have been due to empty rows or columns, which do not influence the number of chips in each piece.)

(There are other ways to check the pieces; for example, we can make one pass through the data, and, for each cell, calculate the number of chocolate chips in the rectangle that has that cell as its lower right corner, and the upper left corner of the waffle as its upper left corner. Then, to find the number of chips in a certain piece, we can add and subtract the appropriate rectangles from this set.)

In summary, this algorithm involves several steps:

1. Create the row and column sum arrays. We can either make two separate passes through every cell of the waffle, or make one pass and create both arrays at once.
2. Convert the sum arrays into cumulative sum arrays.
3. Check the cumulative sum arrays to find our places to cut.
4. Use the cumulative sum arrays to create interval arrays.
5. Make one pass through every cell of the waffle in a directed way, using the interval arrays, to check whether every piece has the same number of chips.

Steps 1 and 5 above involve looking at each of the **R** × **C** cells of the waffle, whereas steps 2, 3, and 4 only involve looking at **R** or **C** values. So steps 1 and 5 dominate the running time, which is O(**R** × **C**). (We could not have done better than this anyway, since we clearly have to look at each cell of the waffle at least once to solve the problem.)

Code Jam 2018 - Round 1A

Analysis: Bit Party

Test set 1

We might consider enumerating all the ways of assigning bits to cashiers, but with 20 bits and 5 cashiers, there could be as many as 5^{20} ways — too many to check! We need to take advantage of the fact that the bits are interchangeable, and instead find all the ways to *partition* **B** bits among **R** of the **C** cashiers (since we will only be able to use as many cashiers as we have robots). Then, we can compute how much time each of those ways takes, and pick the minimum of those values.

We can calculate the number of ways to partition 20 bits among 5 robots using this method. It turns out there are only (24 choose 4) = 10,626 ways to check. If we have fewer robots than cashiers, then we need to introduce another multiplicative factor of (**C** choose **R**), but this cannot be larger than 10 in test set 1. Each check takes O(**R**) time, which is very small given that **R** is at most 5. So, test set 1 should be solvable well within the 15 second time limit, regardless of your language choice.

Test set 2

To solve this test set, we need to be able to answer the following question: Given a time limit T, is there a possible assignment of bits such that all the robots can finish interacting with their cashiers in no more than T seconds? Let f(T) be the answer to this question.

How can we find f(T)? The maximum number of bits that the i-th cashier can process in not more than T seconds is max(0, min($\mathbf{M}_i$, floor((T - $\mathbf{P}_i$) / $\mathbf{S}_i$))). Let us call this value Capacity_i.

Then, we want to know whether a total of **B** bits can be assigned to **R** robots, and each of those robot to a cashier, such that the number of bits processed by the i-th cashier is not more than Capacity_i. To do this, we can greedily sort the Capacity values into nonincreasing order, and then assign the **R** robots to the first **R** cashiers. f(T) is true if and only if the total number of bits that can be processed by the first **R** cashiers is at least **B**. Therefore, we can compute the value of f(T) for any T in O(**C** log(**C**)) time (which is the time it takes to sort the Capacity values). (Aside: We can even avoid the sort, and instead partition in O(**C**) time, by using introselect, for example.)

Since we want to minimize the time taken for all robots to interact with their cashiers, we want to find the minimum possible value of T such that f(T) is true. That value of T will also satisfy the following:

- f(x) is false for all x < T.
- f(x) is true for all x ≥ T.

Therefore, we can find the value of T using binary search. Since the maximum answer will not be more than O(max(**S**) × **B** + max(**P**)), this solution will run in O(**C** log(**C**) log(max(**S**) × **B** + max(**P**))) time.

Code Jam 2018 - Round 1A

Analysis: Edgy Baking

Cookie-cutting

A cookie with width W and height H has a perimeter of 2 × (W + H). If we make a straight cut of length C that divides the cookie in half, each piece will have one side of length C, and the remaining sides of the pieces will represent the perimeter of the original cookie. So, after cutting, we will have a combined perimeter X = 2 × (W + H + C).

Clearly X is smallest when we leave the cookie alone. However, if we want to minimize X given that we are making a cut, we must minimize C; no cut can be smaller than the smallest of W and H, so we should cut through the midpoints of the two larger sides, which gives the cut a size of the smaller of the two sides. Then X = 2 × (W + H + min(W, H)). On the other hand, if we want to make a cut that maximizes X, we must maximize C by cutting through two opposite corners of the cookie. Then C = sqrt(W^2 + H^2), so X = 2 × (W + H + sqrt(W^2 + H^2)). If we start our cut somewhere between one of those larger-side midpoints and a corner, we will get a value of X somewhere between these two extremes; since we can cut anywhere we want in that interval, we can get any intermediate value of X that we want.

So, depending on what we do with our cookie, its contribution to our overall perimeter sum will either be 2 × (W + H), or a value in the range 2 × (W + H) + [2 × min(W, H), 2 × sqrt(W^2 + H^2)]. From now on, we will abbreviate the extremes of that range as [L, R]. (We could divide all values in the problem by 2 as well, removing that factor, but we will retain it in this analysis for clarity.)

Test set 1

In test set 1, all of the cookies have the same dimensions; we will call them W and H. Let us define P' as **P** - 2 × (W + H) — that is, the amount of extra perimeter that we need to add to reach the target **P**. So, we can reformulate the problem as starting with a total perimeter of 0, and trying to reach P'. For each cookie, we can choose to either not cut it, which leaves our total perimeter unchanged, or cut it, which increases our total perimeter by some value of our choice in the range [L, R]. If we cut K cookies, then our total perimeter can be anywhere in the range [K × L, K × R], depending on how we make our cuts. We will think in terms of this range and not in terms of the individual cuts.

How many cookies should we cut? Of course, we should never cut so many that K × L exceeds P'. However, we might as well keep cutting cookies up until that point, since doing so both moves our range closer to the target and makes the range wider. So, we can solve for X in the equation X × L = P', and then choose K to be the floor of X — that is, floor(P' / L). (Since L is an integer representing one of the original side lengths, and P' is also guaranteed to be an integer, we should use integer-based division here to avoid the usual problems with flooring floating-point numbers!)

Then, we can check whether the range [K × L, K × R] includes P'. If it does, the answer is 0; otherwise, it is P' minus the right end of the range, i.e., P' - K × R. Even though R is generally not an integer, we do not need to worry about floating-point issues; even if P' is very close to K × R and we mistakenly conclude that it is not in the range, we will still get an answer that is sufficiently close to 0 under our floating point rules.

Test set 2

In test set 2, different cookies might have different dimensions, so we get different total ranges depending on which ones we cut. We have no a priori basis for deciding which cookies to cut, and we cannot directly check all 2^{100} possibilities, but we can use dynamic programming to ensure that we find the best possibility without explicitly checking them all; the problem is very similar to the knapsack problem. For each cookie, we are deciding whether to leave it as is, or cut it, which adds L to our total perimeter and gives us up to R - L units of additional "slack". Once we have finished deciding which cookies to cut, we can include as much or as little of this "slack" as we want. All other things being equal, having more slack is always better for us. The large limit on **P** may seem daunting, but you can convince yourself that the problem space is actually small enough for methods like this to work.

However, the problem can also be solved in a different way. As in the previous solution, each cookie corresponds to a range [L, R] by which we can increase the total perimeter if we decide to cut that cookie. Let S(K) be the list of intervals we can reach after processing the first K cookies. We start with S(0) = {[0, 0]}.

When we consider the K'th cookie, we can choose to cut it or not. Suppose that this cookie corresponds to the interval [L, R]. If we do not cut it, we can obtain any perimeter that is already in an interval in S(K - 1). If we do cut it, we can reach any interval in the set S' given by S' = {[l + L, r + R] : [l, r] is an interval in S(K - 1)}.

To get S(K), we first take the union of S(K - 1) and S', followed by merging overlapping intervals. For example, if S(K - 1) = {[0, 0], [3, 6]}, and we add the interval [L, R] = [1, 2], we obtain S' = {[1, 2], [4, 8]} and S(K) = {[0, 0], [1, 2], [3, 8]}. Note that the intervals [3, 6] from S(K - 1) and [4, 8] from S' merge into a single interval [3, 8] here.

We can drop any intervals that start after P', so that after processing all N cookies, the answer to the question will be the distance from P' to the last interval.

Without any further analysis, we might expect the size of S(**N**) to be $2^{\mathbf{N}}$, as it could double in each step. It turns out that we can provide a much stronger bound of (log(P') / log(sqrt(2))) + 1.

We first need the observation that any interval [L, R] corresponding to a cookie will satisfy $R \geq \text{sqrt}(2) \times L$, where equality holds for a square. Using induction, we can infer that every interval [l, r] in each S(K) also satisfies $r \geq \text{sqrt}(2) \times l$.

The second observation is that all intervals in S(N) are disjoint. Let us now sort intervals in S(N) and enumerate them starting from $[l_0, r_0]=[0, 0]$. Then the observations together imply that $l_{\{i+1\}} > r_i \geq \text{sqrt}(2) \times l_i$. Since the lower bound of each interval is an integer, we also have $l_1 \geq 1$. Hence we have $l_i \geq \text{sqrt}(2)^{i-1}$. Since we can discard any intervals starting after P', we may assume that $l_i \leq P'$. We conclude that there are at most (log(P') / log(sqrt(2))) + 1 intervals in S(**N**).

This solution takes O(**N** log(P')) time: for each of the **N** cookies, we have to perform one merging step that takes O(|S(K)|) time.

It can also be shown that $|S(K)| \leq 2K + 1$. We leave this as an exercise to the reader!