Solution for Packing Biscuits

We say that a value of y is good if it is possible to pack x bags, each of y total tastiness.

Subtask 1

We proceed by checking every value of y, noting that $y \leq 10^5$. For each value of y, we use the `greedy coin change' algorithm to decide if it is good or not.

Implementing this naively gives a runtime of $\approx 10^5 \cdot x$ operations. There are two ways to speed it up.

One possible way is to remove x coins at a time instead of removing them one by one.

The other way is to check for values of y at most $10^5/x$, so that we have less values of y to check in the even that x is large. However, contestants will have to return 1 immediately if x is larger than 10^5 to avoid actually packing x empty bags!

Subtask 2

We begin with the following observation:

- If $a[i] \geq 3$, the solution remains unchanged if we decrease $a[i]$ by 2 and increase $a[i+1]$ by 1.

Using this observation, we may combine smaller biscuits into bigger ones such that $a[i] \in \{0, 1, 2\}$.

A second observation is the following:

- Suppose $a[i] = 0$ for some i. Let y be any integer, we perform division with remainder and write $y = 2^i \cdot q + r$. Then y is good if and only if $2^i \cdot q$ is good and r is good.
- Suppose $a[i] \neq 0$ for all i. Then for any integer y, y is good if and only if the total tastiness is at least y.

Using this observation, we may split the input into consecutive segments of non-zero values. The final answer is the product of the answers individual segments.

Subtask 3

Similar to the previous subtask, if $a[i] \geq x + 2$, we may decrement $a[i]$ by 2 and increase $a[i+1]$

by 1.

We now proceed by dynamic programming. Let $f(n,i)$ be the answer if we replace $a[0], a[1], \ldots, a[i-1]$ by 0 and increment $a[i]$ by n.

We obtain the following recurrence relation:

$$f(0,60) = 1, f(n,i) = \begin{cases} f(\lfloor \frac{n+a[i]}{2} \rfloor, i+1) & n + a[i] < x \\ f(\lfloor \frac{n+a[i]}{2} \rfloor, i+1) + f(\lfloor \frac{n+a[i]-x}{2} \rfloor, i+1) & n + a[i] \geq x \end{cases}$$

To explain the relation, let S be the set of all valid y when $a[0], a[1], \ldots a[i-1]$ are replaced by 0 and $a[i]$ incremented by n.

If y is a multiple of 2^{i+1}, then the biscuits of tastiness 2^i must be used in pairs. We therefore merge pairs of biscuits to make biscuits of tastiness 2^{i+1}.

If y is not a multiple of 2^{i+1} but not a multiple of 2^i, then we need to use x biscuits of tastiness 2^i. The remaining $a[i] + n - x$ biscuits must be used up in pairs.

If y is not a multiple of 2^i, then y cannot be good.

Subtask 4

Define

$$s[i] = \sum_{j=0}^{i} a[i] \cdot 2^i$$

to be the total tastiness the first $i+1$ types of biscuits.

We need another observation:

- Let $2^i \leq y < 2^{i+1}$. Then y is good if and only if $s[i] \geq x \cdot y$ and $y - 2^i$ is good.

The forward direction is clear. We will justify the backward direction. For simplicity, assume $a[i+1] = a[i+2] = \ldots = a[k-1] = 0$. Since $y - 2^i$ is good, consider some way to pack them, remove these biscuits from our collection.

We need to pack the remaining biscuits into x packs of 2^i each. We do so by merging smaller biscuits into bigger ones- whenever there are two biscuits of tastiness 2^j for some $j < i$, we replace them with a single biscuit of tastiness 2^{j+1}. It can be shown that we will end up with at least x biscuits of tastiness 2^i each after the merging.

We now have an efficient way to enumerate all the solutions. Let A_i be the set of good values which are at most 2^i. We then have

$$A_{i+1} = A_i \cup \{y | y - 2^i \in A_i \text{ and } s[i]/x \geq y\}$$

We can now explicitly list out all elements of A_i. Thus the time taken for a single query is linear in the size of the answer returned.

Subtask 5

Let $g(n)$ be the number of possible y which are less than n. The following recurrence relation solves the problem:

- Let $2^i < n \leq 2^{i+1}$. Then

$$g(n) = g(2^i) + g(\min(n, 1 + s[i]/x) - 2^i)$$

  with the initial values $g(n) = 0$ for all $n \leq 0$ and $g(1) = 1$.

By using a hash table to store all previously computed values, this algorithm runs in $O(k^2)$ time.

The time complexity is justified by the fact that once $g(2^0), g(2^1), g(2^2), \ldots g(2^{i-1})$ are computed, we only need $O(k)$ time to compute $g(2^i)$.