Problem C. Twin Cookies

Subtask 1

You can output what is given in the sample. Alternatively, ask 1, 2 and 3.

Subtask 2

First ask 1 and 2. Let the returned value be x.

Then ask 10 and 20. Let the returned value be y.

Then, ask $y - x$ and $y + x$. If $y - x$ is returned, give x and $y - x$ to the first child, and y to the second. If $y + x$ is returned, give x and y to the first child, and $y + x$ to the second.

Subtask 3

For $i \in \{0, \ldots, 49\}$, first ask $2ni + (1 \ldots n)$, then $2ni + (n + 1 \ldots 2n)$. Let x_i be the first value returned, and y_i the second. Define $d_i = y_i - x_i$.

We have $1 \leq d_i \leq 2n - 1$. Thus, for $n \leq 25$, by the pigeonhole principle, there must exist some pair $i, j \in \{0, \ldots, 49\}$ of distinct indices such that $d_i = d_j$.

Now, give x_i and y_j to the first child, and x_j, y_i to the second.

Subtasks 4-6

The expected solutions to the larger subtasks all employ the same idea, also based on the pigeonhole principle, at differing levels of optimisation.

If we perform k queries, there are 2^k possible subset sums we can create with the k returned values. On the other hand, as long as all values we ask are at most H, the sum of any subset is at most kH. We can achieve $H = kn$ by including the first n unused numbers in every query.

Thus, as long as $2^k > k^2 n$, we can guarantee that some two subsets have the same sum. Even for $n = 5000$, this is achieved for $k \geq 22$.

Given any two different subsets S_1 and S_2 with the same sum, we can now remove $S_1 \cap S_2$ from both. Removing only elements in both sets keeps the sets distinct and their sums equal, so giving the values in $S_1 \setminus (S_1 \cap S_2)$ to the first child, and values in $S_2 \setminus (S_1 \cap S_2)$ to the second is a solution.

It only remains to compute two such subsets S_1 and S_2.

Subtask 4

For subtask 4, we can maintain for every sum $s \leq 101^2 \cdot n$ up to one subset S_s that achieves that sum. After one of our queries gives us the value x to work with, we update these subsets. For every sum s, we see if we can achieve sum $s - x$. If we can achieve that sum, the subset $S_{s-x} \cup \{x\}$ achieves sum s. If the sum s was already achievable, we have found a pair of two different subsets of equal sum. Otherwise, we let $S_s \leftarrow S_{s-x} \cup \{x\}$. Note that we have to make the updates in order of decreasing s, to not create sets that contain multiple copies of x.

Maintaining the subsets as vectors, we do $\mathcal{O}(k \cdot k^2 n \cdot k)$ work. For $k \leq 22$ and $n \leq 200$, this is fast enough.

Subtask 5

Instead of storing the set S_s for every sum s, we store the value x that let us make the set $S_s = S_{s-x} \cup \{x\}$. To actually construct the set at sum s, we can then iteratively add x to the set, and subtract x from s.

Since we do not have to work with vectors, we trim one factor of k, giving a $\mathcal{O}(k \cdot k^2 n)$ algorithm.

Subtask 6

A fast implementation of the $\mathcal{O}(k^3 n)$ algorithm for subtask 5 can pass even subtask 6. However, faster solutions exist as well:

We can still trim one factor of k: notice that as long as no sum is achieved by two different subsets, there are exactly 2^t achievable sums after adding t numbers. Thus, if we maintain a sorted vector C_i of achievable sums at step i, we can compute $C_{i+1} = C_i \bigcup (C_i + x_i)$ in linear time to $|C_i|$ by merging two sorted vectors, and thus we can compute all C_i in $\mathcal{O}(k^2 n)$ work total.

Given an achievable sum s at step i, we can easily compute a set corresponding to it: if $s - x_i \in C_{i-1}$, find the set corresponding to sum $s - x_i$ and step $i - 1$ and add x_i to it, otherwise find the set corresponding to sum s and step $i - 1$.

Given the first sum s and step i that is achievable in two ways, we can then construct the two distinct sets of equal sum by constructing $s - x_i$ and s at step $i - 1$, and adding x_i to the first.

Alternatively, a $\mathcal{O}(k^3 n / \log n)$ solution could be achieved using bitsets. We maintain bitsets B_i with $k^2 n$ bits, with a 1 at position s if the sum s is achievable after adding the first i values x_i we receive (so B_i is a bitset representation of C_i). We have $B_{i+1} = B_i \parallel B_i \ll x_i$, where $\parallel$ is the bitwise OR operation, and $\ll$ the right shift operation. Using AND instead of OR lets us check if any sum can be achieved in two ways. Once we find such a sum and step, we construct the solution as before.

By changing from the first approach to the second after $k - \log \log n$ steps, we could have complexity $\mathcal{O}(k^2 n \log \log n / \log n)$ with the same memory usage, but this is wildly unnecessary for this problem.