

History

Problem author: Ilya Burkov, developer: Andrey Pavlov

Formally, the problem asks us to construct a graph on n vertices that contains all edges (i, j) such that $a_i + a_j = S$ or $a_i \oplus a_j = X$. The answer to the problem is a perfect matching in this graph, and the graph is not necessarily bipartite.

To solve the subtask with $n \leq 20$, one could enumerate all possible partitions into pairs and check whether they satisfy the conditions.

In subtask 3, the graph contains no edges satisfying $a_i + a_j = S$. Therefore, only pairs of the form $(a, a \oplus X)$ are valid. Then it is enough to count cnt_a — the number of occurrences of each value — and check that $\text{cnt}_a = \text{cnt}_{a \oplus X}$ provided that $X > 0$.

If $X = 0$, then all valid pairs satisfying the XOR condition are such that $a_i = a_j$, so for subtask 3 it is enough to check that cnt_a is divisible by 2 for every a . For subtask 4, we again have pairs of the form $a_i + a_j = S$, and we also know how to pair two equal values a , so it is enough to check that $\text{cnt}_a \equiv \text{cnt}_{S-a} \pmod{2}$, while handling the case $2a = S$ separately.

From now on, we will assume that the case $X = 0$ has been handled separately and that $X > 0$.

Consider subtask 6, where all a_i are distinct. Let us understand which vertices are adjacent to vertex i : from the two equations we get that $a_j = S - a_i$ or $a_j = X \oplus a_i$. Since all numbers are distinct, such a j is unique in both cases. Therefore, each vertex i has degree at most 2 in our graph, and sometimes less, because such a j may not exist in the array. Hence, our graph consists of connected components that are simple cycles and paths. For such a graph, a matching can be found constructively using any graph traversal algorithm in $O(n)$.

Now let us move to the full solution. Here each number may occur arbitrarily many times, so we will build our graph a bit differently: we store the same array cnt , keep only distinct numbers in the array, build the same graph, and try to apply the solution for subtask 6. The main difference is that now each vertex additionally has a value cnt written on it. Essentially, for each edge we have an operation that decreases the number on both of its endpoints by 1, and using such operations we need to make all values equal to 0.

For a path, it is enough to understand the following: suppose one endpoint of the path has value x . Then we must take the only edge incident to this endpoint exactly x times. After that, the value at this vertex becomes 0, and at the other endpoint it becomes $y - x$. We can then mentally remove this vertex, after which the path becomes shorter by 1, and we can process the whole path iteratively. If at some point $y - x < 0$, or if one vertex with a nonzero value remains, then no answer exists.

Now suppose we consider a cycle. We can fix any vertex and write out the cycle from it in some direction. Then we could brute-force how many times the first edge of the cycle is taken; after that we can mentally forget about this edge and reduce the problem to the path case. However, this is too slow, so let us optimize it. Formally, suppose the numbers written on the cycle in order are c_1, c_2, \dots, c_k , and we try some $x \leq \min(c_1, c_k)$, after which we consider the path with numbers $c_1 - x, c_2, \dots, c_{k-1}, c_k - x$. Let us see what conditions must hold for this path to be valid: $c_1 \leq c_2$, $c_2 - c_1 \leq c_3$, $c_3 - (c_2 - c_1) \leq c_4$, $c_4 - (c_3 - (c_2 - c_1)) \leq c_5$, ..., and $c_k - (c_{k-1} - (\dots - (c_2 - c_1))) = 0$. Now if we insert the parameter x here, then for every even i we get some expression involving $-x$, while for odd i we get one involving $+x$. Therefore, we obtain $O(k)$ upper and lower bounds on x ; denote them by low and $high$. Then it is enough to take any x from the interval $[low, high]$ that also satisfies the final condition, which is also easy to handle. One implementation issue is the case $2 \cdot a_i = S$, but in this solution it is handled quite simply.

The final implementation can be written using hash maps in $O(l_1 + l_2 + \dots + l_m) = O(n)$, where l_i is the size of the i -th component, which is enough for full score.