Credits: @Peng Cao

Intuition:--------------------------------------------------

Find all prime factors of each array element and construct a graph.

Construct an undirected graph.

Node: Each array element is a node.

Edge: If 2 array elements have at least one common prime factor, then we create an edge connecting them.

However, constructing the graph directly has higher time complexity. We can find all prime factors of each array element, and put the array element into the corresponding lists according to its prime factors.

For example, 60 = 2 ^ 2 \* 3 \* 5

We put 60 in the list of 2, 3, and 5.

Now each pair of elements in the same list should have an edge.

And the key point here is for each list, we just need to create an edge between the first element and any other elements instead.

For example, 3, 15, 60, 99 are in the list of 3.

We only need to create edges (3, 15), (3, 60) and (3, 99).

The question is to ask whether the graph is connected.

Approach:------------------------------------------

Construct the above graph, and check the connectivity.

One way to do it is to use union-find set and check whether it has only one connected component. (Using BFS/DFS is also fine.)

Complexity:-------------------------------------------------------------

Time complexity:

O(sqrt(M) \* N) where M is the largest number (considering each union-find operation takes O(1) time).

Space complexity:

O(N)

-----------------------\_Codes--------------------------------------------------

C++

--------------------

class Solution {

int getf(vector<int> &f, int x) {

return f[x] == x ? x : (f[x] = getf(f, f[x]));

}

void merge(vector<int> &f, vector<int> &num, int x, int y) {

x = getf(f, x);

y = getf(f, y);

if (x == y) {

return;

}

if (num[x] < num[y]) {

swap(x, y);

}

f[y] = x;

num[x] += num[y];

}

public:

bool canTraverseAllPairs(vector<int>& nums) {

const int n = nums.size();

if (n == 1) {

return true;

}

vector<int> f(n), num(n);

for (int i = 0; i < n; ++i) {

f[i] = i;

num[i] = 1;

}

unordered\_map<int, int> have;

for (int i = 0; i < n; ++i) {

int x = nums[i];

if (x == 1) {

return false;

}

for (int d = 2; d \* d <= x; ++d) {

if (x % d == 0) {

if (have.count(d)) {

merge(f, num, i, have[d]);

} else {

have[d] = i;

}

while (x % d == 0) {

x /= d;

}

}

}

if (x > 1) {

if (have.count(x)) {

merge(f, num, i, have[x]);

} else {

have[x] = i;

}

}

}

return num[getf(f, 0)] == n;

}

};

----------------------------------------Java-----------------------------------

class Solution {

private int getf(int[] f, int x) {

return f[x] == x ? x : (f[x] = getf(f, f[x]));

}

private void merge(int[] f, int[] num, int x, int y) {

x = getf(f, x);

y = getf(f, y);

if (x == y) {

return;

}

if (num[x] < num[y]) {

int t = x;

x = y;

y = t;

}

f[y] = x;

num[x] += num[y];

}

public boolean canTraverseAllPairs(int[] nums) {

final int n = nums.length;

if (n == 1) {

return true;

}

int[] f = new int[n], num = new int[n];

for (int i = 0; i < n; ++i) {

f[i] = i;

num[i] = 1;

}

Map<Integer, Integer> have = new HashMap<>();

for (int i = 0; i < n; ++i) {

int x = nums[i];

if (x == 1) {

return false;

}

for (int d = 2; d \* d <= x; ++d) {

if (x % d == 0) {

if (have.containsKey(d)) {

merge(f, num, i, have.get(d));

} else {

have.put(d, i);

}

while (x % d == 0) {

x /= d;

}

}

}

if (x > 1) {

if (have.containsKey(x)) {

merge(f, num, i, have.get(x));

} else {

have.put(x, i);

}

}

}

return num[getf(f, 0)] == n;

}

}