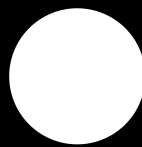




Operation Decoding (Conservation of bits)



Problem Statement: Given an *array* of N positive integers. You can perform this operation any number of times, choose two indices x and y . If $\text{array}[x] = a$ and $\text{array}[y] = b$, then after the operation

1. $\text{array}[x] = a \text{ OR } b$, $\text{array}[y] = a \text{ AND } b$.

Perform the operations optimally such that $\sum \text{array}[i]^2$ for all $1 \leq i \leq n$ is maximum. Print the largest sum of squares you can get after performing the operations greater than equal to zero times.

Explanation:

The given algorithm aims to maximize the sum of squares of an array by performing specific operations on the array elements. The allowed operations involve choosing two indices, (x) and (y) , and updating array values according to the rules:

$\text{arr}[x] = \text{arr}[x] \mid \text{arr}[y]$,
 $\text{arr}[y] = \text{arr}[x] \& \text{arr}[y]$

The algorithm uses a greedy approach to maximize the sum of squares. It iterates through the bits of the numbers in descending order, starting from the most significant

bit. For each bit, it identifies the element with the bit set in its binary representation and tries to increase its value as much as possible.


A[4] = 1,3,5,6

	1	3	5	6			
L3	0	0	1	1	After several	1	1
L2	0	1	0	1	==>	0	0
L1	1	1	1	0	operations	1	1

In each level, no. of one will be constant if travelling across "in a particular" level.

Code:

```
vector<ll> bit[20];
ll n;
cin >> n;
vector<ll> arr(n);
for (int i = 0; i < n; i++)
    cin >> arr[i];
vector<ll> done(n + 1, 0);
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < 20; j++)
    {
        if (arr[i] & (1 << j))
            bit[j].push_back(i); // push in bit[i] all the index which have it
    }
}
ll k = 19;
while (k >= 0)
{ // Greedily increasing the value of every number, starting from the most sig
    if (bit[k].size())
    {
        ll x = bit[k][bit[k].size() - 1]; // taking a element which may or may
        bit[k].pop_back();
        if (done[x] or !((1 << k) & arr[x])) // if it is already done or bit i
            continue;
        done[x] = 1; // trying to increase the value as much as I can using th
        for (int i = 0; i < k + 1; i++)
        {
            if (arr[x] & (1 << i)) // it bit is already set then continue
                continue;
            while (bit[i].size())
            { // find a index which has this bet set, then apply the operation
                ll y = bit[i][bit[i].size() - 1];
                bit[i].pop_back();
                if (done[y] or !((1 << i) & arr[y]))
```

 Copy

```
        continue;

        ll temp = arr[y];
        arr[y] = arr[y] & arr[x]; // because of this operation the set
        arr[x] |= temp;
        break;
    }
}
else
    k--;
}
ll ans = 0;
for (int i = 0; i < n; i++)
    ans += arr[i] * arr[i];
cout << ans << "\n";
```

Time Complexity:

The time complexity of the algorithm is primarily determined by the number of bits in the binary representation of the array elements. The outer loop iterates through each bit, and for each bit, the algorithm may perform operations on elements with that bit set. In the worst case, the time complexity is $O(20 * n)$, where 'n' is the number of elements in the array and 20 is the assumed maximum number of bits in the binary representation of array elements. However, in practice, the number of iterations is often much smaller, leading to an efficient algorithm in most scenarios.

Mark as Read ✓

Write a comment

B

I

U

</>

🔗