Rating - Editorial - I try to show not just the solution but also how to arrive at it

Timothy Gao

We are given an array of size N ($1 \leq N \leq 1500$), and in one move you can choose any two distinct element left in the array a and b, add $a^b \mod 10^9+7$ or $b^a \mod 10^9+7$ to the sum, and remove one of the numbers a or b from the array. Return the maximum sum we can obtain after some number of operations, modulo 10^9+7.

1 Initial Observations

If we imagine there are no deletions, then the solution is intuitive: we find the pair that yields the maximum result after one operation, and keep choosing those numbers. Let's we visualize the array as a graph. The nodes are the elements of the array. We draw undirected edges between every pair of nodes (i,j), with weight of $max(val_i^{val_j} \mod 10^9+7, val_j^{val_i} \mod 10^9+7)$. (Fun fact: we originally intended for edge weights of only $max(val_i^{val_j}, val_j^{val_i})$, without the mod, which requires comparing the quantities $\frac{val_i}{\ln val_i}$ and $\frac{val_j}{\ln val_j}$, but we thought this would be unclear to some contestants.) Essentially, if there were no deletions, we would be choosing the same edge $N-1$ times.

This provides us with some insight in how we can formulate the problem. Given that there are deletions, we can never choose the same edge twice, as one node of the two must be discarded after the first time we choose it. In fact, one node of the two can never be used in another edge again. Following this logic, the edges we choose will never form a cycle because this requires every node to be part of exactly two edges (accessed twice). Alternatively, notice that in just the first operation in the cycle alone (the first edge drawn in the cycle), at least one node must be discarded, meaning that that node can only be operated on once. Because we end up with a single node (element in the array), the entire graph must be connected by edges. A connected graph without cycles implies that any choice of edges must form a tree. Alternatively, notice that a total of exactly $N-1$ edges will be drawn, because exactly $N-1$ removes can be applied before there is only one node left. A connected graph with $N-1$ edges once again implies a tree, further confirming our previous result.

2 Arriving at the Solution

To attain the maximum sum of edges, the problem reduces to finding the maximum spanning tree in the graph. The most straightforward way to do this is to generate a graph with the nodes and edges as aforementioned and apply Kruskal's or Prim's onto this generated graph. Each of the N^2 edges can be pre-calculated in $log(N)$ via binary exponentiation, for a $O(N^2 log(N))$ pre-calculation. Then, the MST algorithm itself will run with the same time complexity, leading to a total time complexity of $O(N^2 log(N))$ which is sufficient for the time limit.

We can improve this by modifying Prim's algorithm for Dense Graphs, which allows us to speed up the solution by a constant factor. The idea is to keep an array *bestDist* as the maximum distance of all edges between nodes included and excluded in the spanning tree. Each time we insert a new node into our spanning tree, we can calculate its distance to all remaining free nodes (nodes not in our spanning tree) on the fly, updating *bestDist* and keeping track of the maximum distance free node that we find (which is the one to include the next iteration). With binary exponentiation, this algorithm also runs in $O(N^2 log(N))$, but with a significantly lower constant factor. However, this optimization is not required to obtain full credit for the problem.

3 Code

3.1 With Kruskal's Algorithm

```
//@timothyg

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;

#define pb push_back

#define f first
#define s second

struct DSU{
    vector<int> par, sz;
    DSU(int n){
        par.reserve(n); sz.reserve(n);
        for(int i = 0; i<n; i++) {
            par.pb(i); sz.pb(1);
        }
    }

    int root(int u){
        return par[u] == u ? u : par[u] = root(par[u]);
    }

    void unite(int u, int v){
        int ru = root(u); int rv = root(v);
        if(ru == rv) return;
        if(sz[rv] > sz[ru]) swap(ru, rv);
        sz[ru] += sz[rv];
        par[rv] = ru;
    }

};

const int mod = 1e9 + 7;

int binpow(ll base, ll pow)
{
    ll ret = 1;
    while(pow > 0){
        if(pow & 1) ret = (ret * base)%mod;
        base = (base * base)%mod;
        pow >>= 1;
    }
    return ret;
}

ll get(ll a, ll b)
{
    return max(binpow(a, b), binpow(b, a));
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(0);
    int N; cin >> N;
    vector<int> nums(N);
    for(int i = 0; i<N; i++) cin >> nums[i];
    vector<array<int, 3>> edges;
    for(int i = 0; i<N; i++){
        for(int j = 1; j<N; j++){
            int best = get(nums[i], nums[j]);
            edges.pb({-best, i, j});
        }
    }
    ll ret = 0;
    DSU dsu(N);
    sort(edges.begin(), edges.end());
    for(auto i : edges){
        if(dsu.root(i[1]) == dsu.root(i[2])) continue;
        ret += -i[0];
        ret %= mod;
        dsu.unite(i[1], i[2]);
    }
    cout << ret << '\n;
    return 0;
}
```

3.2 With Prim's Algorithm with Dense Graph Optimization

```
//@timothyg

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;

#define pb push_back

#define f first
#define s second

const int mod = 1e9 + 7;

int binpow(ll base, ll pow)
{
    ll ret = 1;
    while(pow > 0){
        if(pow & 1) ret = (ret * base)%mod;
        base = (base * base)%mod;
        pow >>= 1;
    }
    return ret;
}

ll get(ll a, ll b)
{
    return max(binpow(a, b), binpow(b, a));
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    int N; cin >> N;
    vector<ll> nums(N);
    for (int i = 0; i < N; i++) cin >> nums[i];

    ll ret = 0;
    vector<ll> inTree(N), Dist(N);
    //InTree is a 0/1 vector representing wether the ith node is in the spanning tree
    //Dist is bestDist as described in the editorial

    inTree[0] = 1;
    for(int i = 1; i<N; i++){
        ll best = max(get(nums[i], nums[0]), get(nums[0], nums[i]));
        Dist[i] = best;
    }
    for (int i = 1; i < N; i++)
    {
        int j = -1;
        for (int k = 0; k < N; k++)
        {
            if (inTree[k])
                continue;
            if (j == -1 || Dist[k] > Dist[j])
            {
                j = k;
            }
        }
        //calculate new approximate distances
        ret += Dist[j];
        ret %= mod;
        inTree[j] = 1;
        for (int k = 0; k < N; k++)
        {
            if(inTree[k]) continue;
            ll best = max(get(nums[j], nums[k]), get(nums[k], nums[j]));
            Dist[k] = max(Dist[k], best);
        }
    }
    cout << ret << '\n;
    return 0;
}
```