TJIOI 2022 Editorial

TJ Computer Teams

June 2022

Preface

We hoped everyone enjoyed the problems! TJIOI will be back next year in June.

Contents

Problem A: 3rd Floor Pool

This problem can be brute forced because of the relatively small size of N. First, observe that the sum of the first two elements must equal the sum of the remaining elements, minus the third element. Thus, we can brute force which two elements we select as the first two. Simply loop over all pairs of indices, i and j such that $i \neq j \neq 3$, and check if $a_i + a_j = \frac{tot}{2}$, where tot is the sum of all elements in the array. This ensures that the sum of the first two elements, is the same as the sum of the remaining elements. The time complexity, thus, will end up being $O(N^2)$.

```
int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    int t; cin>>t;
    while(t--){
        int n; cin>>n;
        int ar[n];
        for(int i=0; i<n; i++) cin>>ar[i];
        int sm = 0;
        for(int i=0; i<n; i++) sm += ar[i];
        for(int i=0; i<n; i++){
            for(int j=i+1; j<n; j++){
                if(i == 2 || j == 2) continue;
                if(2 * (ar[i] + ar[j]) == sm){
                    cout<<ar[i]<<" "<<ar[j]<<" "<<0<<" ";
                    for(int l=0; l<n; l++) if(l != 2 && l != i && l !=
                        j) cout<<ar[l]<<" ";
                    nl;
                    goto skip;
                }
            }
        }
        cout<<-1<<"\n";
        skip:;
    }
}
```

Problem Idea: Johnny Liu
Prepared By: Johnny Liu

Problem B: Reverse Addition

For this problem, we should tackle the ranges that start from even indices separately from ranges that start from odd indices. We can then iterate through the array and keep track of two sums – one for the reverse addition sum from position 0 to the current index, and one from position 1 to the current index. We also keep track of the minimum of these sums, for the even and odd ranges. Lastly, we just need to take the minimum sum and subtract it from the current sum in order to find the largest reverse addition sum that ends on the index we are checking. Again, we will have to compute this value separately for ranges that begin on even and odd indices. Our answer will be the maximum of these differences.

```
int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n; cin>>n;
    ll sm[2] = {0, 0};
    ll mn[2] = {(ll)2e18, 0};
    ll ans = -2e18;
    for(int i=0; i<n; i++){
        ll x; cin>>x;
        if(i) sm[i%2] -= x;
        sm[1-(i%2)] += x;
        if(i) ans = max(ans, sm[i%2] - mn[i%2]);
        mn[(i%2)] = min(mn[(i%2)], sm[(i%2)]);
    }
    cout<<ans;
}
```

Problem Idea: Kevin Shan
Prepared By: Kevin Shan

Problem C: Climbing Mountains

The main idea is to find the mountains whose peaks are not "covered" by other mountains. Then, locate intersections between the mountain peaks. The area can be computed very easily afterwards. We will utilize a stack, and iterate through mountains in increasing order of the x-coordinates of their peaks. All mountains in the stack will have peaks that are not covered. One property of the peaks is that if a peak with x-coordinate x_1 is fully exposed, and another peak with x-coordinate x_2 is fully exposed, and $x_1 < x_2$, all peaks with x-coordinate $x > x_2$ which are not covered by the second mountain will also not be covered by the first mountain. Using this property, when checking whether or not to add a mountain to the stack, we keep popping from the top of the stack until the current mountain does not cover the peak of the mountain at the top of the stack. Then, if the current mountain is not covered by the top of the stack, we add it to the stack. Some geometry, utilizing the fact that the slopes of the mountains are 1 and -1, will allow us to check for this covering. As for calculating the area, realize that the area contained between two points is the average of the two points, times the distance between them.

Explanation on finding the intersection of two mountains: Given two mountain peaks with coordinates (x_1, y_1) and x_2, y_2 and $x_1 < x_2$, define d to be the value where $x_1 + d$ is the x-coordinate of the intersection. Additionally, define the distance between the mountains to be $L = x_2 - x_1$. Because the mountains slopes have slopes of -1 and 1 respectively, at point x_1, the first mountain has y-coordinate $y_1 - d$. Likewise, the y-coordinate of the second mountain is $y_2 - (L - d)$. We now have the following equation:

$$y_1 - d = y_2 - (L - d)$$

$$d = \frac{y_1 - y_2 + L}{2}$$

We can now use this value of d to find the area, by using the averaging principle described earlier.

```
#define f first
#define s second

int main() // modulus operations not included for readability
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    ll n, L; cin>>n>>L;
    L *= 2;
    vector<pair<ll, ll>> ar;
    for(int i=0; i<n; i++){
        ll x, h; cin>>x>>h;
        x *= 2;
        h *= 2;
        ar.push_back({x, h});
    }
    sort(all(ar));
    vector<pair<ll, ll>> v;
    v.push_back({0, -2e18});
    for(auto p:ar)
        v[0].s = max(v[0].s, p.s - p.f);
    for(auto p:ar){
        if(v.size() == 1) v.push_back(p);
        else{
            while(v.size()>1 && v.back().s + p.f - v.back().f < p.s)
                v.pop_back();
```

```
            if(v.size()>1 && v.back().s - (p.f - v.back().f) >= p.s)
                continue;
            v.push_back(p);
        }
    }
    ll ans = 0;
    v.push_back({L, v.back().s - L + v.back().f});
    for(int i=1; i<v.size(); i++){
        ll a = v[i-1].s;
        ll b = v[i].s;
        ll x = v[i].f - v[i-1].f;
        ll d = (a - b + x) / 2;
        ans += d * (2 * a - d) + (x-d) * (2 * b - (x-d));
    }
    assert(ans % 2 == 0);
    cout<<ans/2;
}
```

Problem Idea: Kevin Shan
Prepared By: Kevin Shan

Problem D: Buzzer Beater

Kevin's team can only make a pass to a teammate if his team can receive the ball before Arnav's team can reach that teammate. We can perform two Dijkstra's. The first one will find the shortest amount of time it will take any given teammate of Kevin's to be reached by one of Arnav's players. The second Dijkstra's will calculate the minimum time required for Kevin's team to make passes do a teammate. For the second Dijkstra's, we will need an additional state to tell us how many teammates have touched the ball, which is stored in $dp[node][players]$. We take any state with the largest $players$ parameter, and return $players$. Alternatively, because the graph is acyclic, we can use topological sort and dynamic programming to solve the remainder of the problem in a similar fashion. See code for implementation details.

```
int ar[MAXN];
ll dp[MAXN][MAXN];
ll dst[MAXN];
vector<pair<int, ll>> g1[MAXN];
vector<int> g2[MAXN];

struct node{
    ll w; int x, d;
    friend bool operator<(node a, node b){
        return a.w > b.w;
    }
};

int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, m; cin>>n>>m;
    int Q; cin>>Q;
    for(int i=0; i<n; i++){
        for(int j=0; j<=n; j++){
            dp[i][j] = 2e18;
        }
        dst[i] = 2e18;
    }
    for(int i=0; i<n; i++){
        cin>>ar[i];
    }
    for(int i=0; i<m; i++){
        int u, v; ll x;
        cin>>u>>v>>x;
        u--; v--;
        g2[u].push_back(v);
        g1[u].push_back({v, x});
        g1[v].push_back({u, x});
    }
    priority_queue<pair<ll, int>, vector<pair<ll, int>>,
        greater<pair<ll, int>>> q1;
    priority_queue<node> q2;
    while(Q--){
        ll t; int p; cin>>t>>p;
        p--;
        dst[p] = min(dst[p], (ll)t);
        q1.push({dst[p], p});
    }
    while(q1.size()){
        auto c = q1.top(); q1.pop();
        if(c.f > dst[c.s]) continue;
```

```
        for(auto e : g1[c.s]){
            ll nw = c.f + e.s;
            if(nw < dst[e.f]){
                dst[e.f] = nw;
                q1.push({nw, e.f});
            }
        }
    }
    dp[0][1] = 0;
    q2.push({0, 0, 1});
    int ans = 1;
    while(q2.size()){
        node c = q2.top(); q2.pop();
        ans = max(ans, c.d);
        for(int a : g2[c.x]){
            ll nw = c.w + ar[c.x];
            if(nw < dp[a][c.d + 1] && nw < dst[a]){
                dp[a][c.d + 1] = nw;
                q2.push({nw, a, c.d + 1});
            }
        }
    }
    cout<<ans;
}
```

Problem Idea: Jesse Choe
Prepared By: Jesse Choe

Problem E: Candyland

Define a location as a node, and a path as an edge. The graph is a binary tree, with the exception of the root which can have 3 edges. We are asked to find the sum of the values for all of the nodes, which means we have to somehow query each node. A query finds the sum of all the nodes on the path between two nodes, so it can be seen that it is optimal to query from leaf nodes. In this way, each query can at most eliminate two leaf nodes, and it is always optimal to do this, as you want to eliminate leaf nodes as fast as possible.

To decide which leaf nodes to pair together, a factor that has to be considered is whether or not the path splits the tree. Since each node can only be visited by a query, once a path is taken, the tree is broken into parts, and no nodes on either side of it can be paired together anymore. Given two leaves A and B, any leaves that are closer to A than B will be split off, and vice versa for B. To prevent splitting, each leaf should be paired with the closest leaf to it.

The above process is repeated whenever the tree splits, or whenever a node has two children. The root node can be dealth with separately, if it has 3 children, by just using an additional operation for the last strip that remains.

Implementation can be done with a dfs: perform an operation if a node has two children. Recursively call on the children to essentially trim the tree wherever there is a split.

```
int N,Q;
V<vi> adj, path;

bool dfs(int x, int p) {
    vi children;
    for (int y: adj.at(x)) if (y != p && dfs(y,x)) {
        children.push_back(y);
    }
    if (children.size() > 2) children.resize(2);
    for (int c: children) {
        path.at(x).push_back(c), path.at(c).push_back(x);
    }
    return children.size() < 2;
}

int main() {
    setIO();
    re(N,Q);
    adj.resize(N);
    path.resize(N);
    rep(N-1) {
        int a, b; cin>>a>>b;
        adj[a].push_back(b), adj[b].push_back(a);
    }
    dfs(0,-1);
    vb vis(N);
    int ans = 0;
    FOR(i,N) if (!vis[i] && path[i].size() <= 1) {
        int cur = i;
        while (true) {
            vis[cur] = true;
            if (sz(path[cur]) == 0) break;
            assert(sz(path[cur]) == 1);
```

```
            int nex = path[cur][0];
            path[nex].erase(find(all(path[nex]),cur));
            cur = nex;
        }
        cout<<"? "<<i<<" "<<cur<<endl;
        int res; cin>>res;
        ans += res;
    }
    cout<<"! "<<ans<<endl;
}
```

Problem Idea: Johnny Liu
Prepared By: Johnny Liu

Problem F: Falling

Firstly, let's find the probability of the ball rolling off each side of a platform with endpoints (l, r) when it is at position x. Let us find the probability of the ball rolling off the left side, $P_l(l-1) = 1$, as the ball has already rolled off. Likewise, $P_l(r+1) = 0$ as the ball has rolled off the right side. To find the other positions, it can be seen that

$$P_l(x) = \frac{1}{2}P_l(x-1) + \frac{1}{2}P_l(x+1),$$

or

$$P_l(x) = \frac{P_l(x-1) + P_l(x+1)}{2}.$$

Because at each position, the probability of reaching such a position is the average of the neighboring two probabilities, we are left with an arithmetic sequence. So,

$$P_l(x) = \frac{r+1-x}{r-l+2}.$$

Likewise,

$$P_r(x) = \frac{x-l+1}{r-l+2}.$$

We can approach the remainder of the problem through dynamic programming. First, sort all platforms by decreasing y-coordinate, then loop through the platforms by y-coordinate. Let $dp[x]$ be the probability of a ball being at position x at whatever y-coordinate currently being iterated on. For each platform at this level, we iterate through the x-coordinates, fron l to r. Because of the small size of the platforms, this means we will end up iterating through a maximum of $N * 12$ distinct x-coordinates. At each x-coordinate, x, we perform the following operations using the previous P_l and P_r functions we found earlier:

$$dp[l-1] = dp[l-1] + dp[x] \cdot P_l(x)$$
$$dp[r+1] = dp[r+1] + dp[x] \cdot P_r(x)$$
$$dp[x] = 0.$$

$dp[x]$ is set to zero as the ball cannot pass through the platform, so the probability is 0. We can store the dp array with a map. Whever an x-coordinate is accessed, which is not already present in the map, we can set $dp[x] = 1/L$ as the ball is dropped at any point from 1 to L with equal probability.

```
ll mul(ll a, ll b){ return (a * b) % MOD; }
ll fpow(ll b, ll p){
    ll r = 1;
    for(; p; p>>=1, b=mul(b, b)) if(p&1) r=mul(r, b);
    return r;
}
ll minv(ll a){ return fpow(a, MOD-2); }

struct pf{
    int h, l, r;
    friend bool operator<(pf a, pf b){
        return a.h > b.h;
    }
};

int main()
```

```
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, q; ll L;
    cin>>n>>q>>L;
    vector<pf> ar;
    for(int i=0; i<n; i++){
        int h, l, r;
        cin>>l>>r>>h;
        ar.push_back({h, l, r});
    }
    sort(all(ar));
    int p = 0;
    map<int, ll> dp;
    while(p < n){
        int y = ar[p].h;
        while(p < n && ar[p].h == y){
            int l = ar[p].l;
            int r = ar[p].r;
            if(dp.count(l-1) == 0) dp[l-1] = minv(L);
            if(dp.count(r+1) == 0) dp[r+1] = minv(L);
            for(int i=l; i<=r; i++){
                if(dp.count(i) == 0) dp[i] = minv(L);
                dp[l-1] += (r+1-i) * minv(r-l+2) % MOD * dp[i] % MOD;
                dp[r+1] += (i-l+1) * minv(r-l+2) % MOD * dp[i] % MOD;
                dp[l-1] %= MOD;
                dp[r+1] %= MOD;
                dp[i] = 0;
            }
            p++;
        }
    }
    while(q--){
        int p; cin>>p;
        if(dp.count(p)) cout<<dp[p]<<"\n";
        else cout<<minv(L)<<"\n";
    }
}
```

Problem Idea: Kevin Shan
Prepared By: Kevin Shan

Problem G: Divisible

Firstly, consider what it means for a sum to be divisible by K. We simplify this condition to be, given a subset, it will satisfy the condition if $sum \mod K = 0$.

The key observation for that problem is that K is rather small. Because we can negate values, one possible strategy to end up with a sum divisible by K is to find two disjoint subsets with the same positive sum. Assume all subsets of the array have distinct sums: when $N \geq 20$, there will be more than 10^6 distinct sums. Because we are modding by K, and $K \leq 10^6$, by pigeonhole principle, it is guaranteed that two of these subsets will share the same sum.

The last part of the problem is constructing the two subsets. If we have found two subsets which share the same sum, it is not enough to simply print out one such subset as the positive values, and the other as negative values, because they could possibly share elements. To solve this, we can just omit indices which are shared by both subsets.

It suffices to just use the first 20 elements to solve this problem, if $N > 20$. Otherwise, iterate through the subsets of all N elements. This can be done through iterating on bitmasks. The time complextity ends up being $O(2^{\min(N,20)})$.

```
int ar[MAXN];

int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, k; cin>>n>>k;
    for(int i=0; i<n; i++){
        cin>>ar[i];
    }
    n = min(n, 20);
    map<ll, int> esm;
    for(int m1 = 0; m1 < (1<<n); m1++){
        ll sm = 0;
        for(int i=0; i<n; i++) if((1<<i) & m1) sm += (ll)ar[i];
        sm %= k;
        if(esm.count(sm)){
            int m2 = esm[sm];
            vector<int> ans[2];
            for(int j=0; j<n; j++){
                int b = (1<<j);
                if((b&m1) && (b&m2)) continue;
                if(b&m1) ans[0].push_back(j+1);
                if(b&m2) ans[1].push_back(j+1);
            }
            cout<<"YES\n";
            cout<<ans[0].size()<<" ";
            for(int x : ans[0]) cout<<x<<" ";
            cout<<"\n";
            cout<<ans[1].size()<<" ";
            for(int x : ans[1]) cout<<x<<" ";
            return 0;
        }
        esm[sm] = m1;
    }
    cout<<"NO";
}
```

Problem Idea: Ray Bai
Prepared By: Kevin Shan

Problem G: Matching

(Fun Fact: The original intended solution to this problem was incorrect, so this problem ended being more difficult than it was made to be.)

There are multiple solutions to this problem, including one using square root decomposition. The (not so) intended solution detailed here uses another method of optimization.

The problem simplifies down to finding a subset of elements such that, when sorted, $a_0 \cdot 2 \leq a_1, a_1 \cdot 2 \leq a_3, ...$ The crucial observation to this problem is that a valid subset can contain at most $\log(a_{max})$ items, as the elements must at least double for each additional element. So, the following algorithm can be used to answer the queries:

1. For each query, greedily select the smallest element in the range to begin with. Call this element x.
2. Locate, in the range, the smallest element that is at least $2 \cdot x$. This element can now be represented with x.
3. Repeat the above step, until there are no more elements that are satisfy the condition.

It turns out that this greedy strategy works, because it is always optimal to select the smallest element possible to add to the subset. However, finding the smallest element that is at least $2x$ is not so easy. Thus, we can approach the problem with this strategy in mind, and instead, solve all the queries simultaneously.

Notice that $a_{max} \leq 10^4$, which is relatively small. We iterate on i in increasing order of a_i. For each a_i, we wish update all queries that have a_i as the smallest available element that is at least $2x$, as established before.

To set up this process, first find the smallest element in each query. This can be done with an RMQ data structure, which takes $O(N \log N)$ time complexity to compute, and $O(1)$ for each query. We will store queries in a segment-tree-like data structure, which will essentially break down ranges into at most $\log(N)$ chunks. Each node of the segment tree will store a set of the indices of all queries which are contained within that interval. For each initial query, add it to a list of query indices at $seg[2x]$.

Now, when we loop through each element in increasing order, we know that we will be at the minimum a_i that has been not visited yet. For this value of a_i, we want to update as many queries as we can, as we would like to use the minimum valid value for each query. To do this, we query i in our segment tree, and return all the indices of queries which contain i. Then, we update the segment tree and remove all of these queries. For each query that was removed, add it to the list at $seg[2 \cdot a_i]$, to set up for the next item in the subset. We can increment the answer count for each of these queries as well, indicating that we just added another item to each subset.

This above process works, as we are essentially pushing forward any unsatisfied queries. Because we are iterating on elements on increasing order, we will always find the optimal subset for each query. See the code below for implementation details of the segment tree.

The time complexity for this problem is $O(N \log N + Q \cdot \log(N) \cdot log(a_{max}) \cdot log(Q))$.

- $O(Q)$ for simultaneously processing each query
- $O(\log(a_{max}))$ for the maximal length of each subset
- $O(\log(Q))$ for the removing and inserting of query indices in the sets of each node of the segment tree
- $O(\log(N)$ for the segment tree structure

```
set<int> intervals[4 * MAXN];
vector<int> out;

void query(int x, int l, int r, int v){
    if(l > r || x > r || x < l) return;
    for(int i:intervals[v]) out.push_back(i);
    if(x == l && x == r) return;
    int m = (l+r)/2;
    query(x, l, m, v*2);
    query(x, m+1, r, v*2+1);
}
void change(int v, int flg, int x){
    if(flg == 1) intervals[v].insert(x);
    else intervals[v].erase(intervals[v].find(x));
}
void update(int x, int l, int r, int tl, int tr, int v, int flg){
    // tl, tr = true left, right
    if(l > r || l > tr || r < tl) return;
    else if(tl <= l && tr >= r) {
        change(v, flg, x);
    }
    else{
        int m = (l+r)/2;
        update(x, l, m, tl, tr, v*2, flg);
        update(x, m+1, r, tl, tr, v*2+1, flg);
    }
}

int rmq[MAXN][21];
int lg[MAXN];
list<int> pos[INF];
int rng[MAXN][2];
list<int> seg[INF];

int mnq(int l, int r){
    int z = lg[r-l+1];
    return min(rmq[l][z], rmq[r+1-(1<<z)][z]);
}

int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0);
    int n, q; cin>>n>>q;
    lg[1] = 0;
    for (int i=2; i<=n; i++) lg[i] = lg[i/2] + 1;
    int ar[n];
    for(int i=0; i<n; i++) {
        cin>>ar[i];
        pos[ar[i]].push_back(i);
    }
    for(int k=0; k<21; k++){
        for(int i=0; i<n; i++){
```

```
            if(k == 0) rmq[i][k] = ar[i];
            else if(i + (1<<k) <= n) rmq[i][k] = min(rmq[i][k-1], rmq[i
                + (1<<(k-1))][k-1]);
        }
    }
    vector<int> ans(q);
    for(int i=0; i<q; i++){
        cin>>rng[i][0]>>rng[i][1];
        rng[i][0]--; rng[i][1]--;
        int nxt = 2*mnq(rng[i][0], rng[i][1]);
        if(nxt < INF) seg[nxt].push_back(i);
    }
    for(int i=1; i<INF; i++){
        for(int j:seg[i]){
            update(j, 0, n-1, rng[j][0], rng[j][1], 1, 1);
        }

        for(int j:pos[i]){
            out.clear();
            query(j, 0, n-1, 1);
            for(int l: out){
                ans[l]++;
                update(l, 0, n-1, rng[l][0], rng[l][1], 1, -1);
                if(2*i < INF) seg[2*i].push_back(l);
            }
        }
    }
    for(int i:ans) cout<<i+1<<"\n";
}
```

Problem Idea: Andrew Wang, Kevin Shan
Prepared By: Kevin Shan