Blitz 2 Editorial

June 8th, 2020

1 Omkar and Explosions

To determine if the product exceeds 10^9, process the numbers while keeping a running product. To prevent overflow, use long integers and terminate as soon as your number exceeds 10^9. Be careful when one of the numbers is 0, as that will make the entire product 0, regardless of how big the other numbers are.

Time Complexity: $O(N)$

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n; cin >> n;
    vector<long long> nums(n);
    for(auto &e : nums) cin >> e;
    sort(nums.begin(), nums.end());
    if(!nums[0]) { cout << "0" << endl; return 0; }
    long long run = 1;
    for(auto &e : nums) {
        run *= e;
        if(run > 1e9) { cout << "EXPLOSION!!!" << endl; return 0; }
    }
    cout << run << endl;
}
```

2 Omkar and Salary

A brute force approach of finding every elder for each Omkaryote runs in $O(N^2)$, which is far too slow given the input constraints. Instead, sort the Omkaryotes by rank from highest to lowest and iterate through them, keeping track of how many have already been processed. The amount of money an Omkaryote will give out in this ordering is $x_i \cdot (\#\ processed)$. Be careful to avoid giving money to people of the same rank, as elders must have a **strictly** greater rank.
Time Complexity: $O(N \cdot log_2 N)$

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n; cin >> n;
    vector<pair<int, long long>> omk;
    for(int i = 0; i < n; i++) {
        int a, b; cin >> a >> b;
        omk.push_back({a, b});
    }
    sort(omk.begin(), omk.end(), greater<pair<int, long long>>());
    long long ans = 0, cur = 0, count = 0;
    for(int i = 0; i < n; i++) {
        if(i and omk[i].first != omk[i-1].first) { count += cur; cur = 0; }
        ans += omk[i].second*count;
        cur++;
    }
    cout << ans << endl;
}
```

3 Omkar and Test

A good first step is to do a prefix sum on the changes to get the actual test scores on each date. Iterate through the test scores, keeping track of a running minimum. Assume that every day you process will be the second test day, and the minimum thus far was the first day. Take the maximum difference over all possible test dates. This question equates to the maximum subarray sum question.
Time Complexity: $O(N)$

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n; cin >> n;
    vector<long long> scores = {0};
    long long ans = INT_MIN, run = 0;
    for(int i = 0; i < n-1; i++) {
        long long c; cin >> c;
        scores.push_back(scores.back()+c);
        ans = max(ans, scores.back()-run);
        run = min(run, scores.back());
    }
    cout << ans << endl;
}
```

4 Omkar and Race

This question can be solved using dynamic programming, where a state $dp[i][j]$ represents the least number of steps needed to reach platform i with j potion uses left. Transitions would be either moving by B_1, B_2 or using the potion. However, due to the large size of $N \cdot K$, this solution runs out of both time and memory. The code can be optimized by noting that even though up to K potions can be used, this number is at most $min(K, log_2 N)$, since each potion doubles the current position.
Time Complexity: $O(N \cdot min(K, log_2 N))$

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

int dp[200001][31];

int main() {
    int n, k, a, b; cin >> n >> k >> a >> b;
    k = min(k, 30);
    for(int i = 0; i <= n; i++)
        for(int j = 0; j <= k; j++) dp[i][j] = 1e9;
    dp[1][k] = 0;
    for(int i = 1; i <= n; i++) for(int j = 0; j <= k; j++) {
        dp[i+a][j] = min(dp[i+a][j], dp[i][j]+1);
        dp[i+b][j] = min(dp[i+b][j], dp[i][j]+1);
        if(j) dp[i*2][j-1] = min(dp[i*2][j-1], dp[i][j]+1);
    }
    int ans = 1e9;
    for(int i = 0; i <= k; i++) ans = min(ans, dp[n][i]);
    cout << (ans == 1e9 ? -1 : ans) << endl;

}
```

5 Omkar and Flex

Imagine plotting each Omkaryote on the coordinate plane, with the x-axis being age, and y-axis being experience. The question then becomes a query for how many points are in the lower left quadrant for each of the points. Run a sweep of the points from lowest experience to highest experience, using a data structure to keep track of the ages. At every point, query on the data structure for how many processed points are at most d_i below in age. A Fenwick tree or segment tree work as possible data structures. Be careful not to count points which have the same experience level.
Time Complexity: $O(N \cdot log_2 N)$

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

struct Fenwick {
    vector<int> t;
    Fenwick(int sz) { t.resize(sz+1); }
    int value(int i) {
        int s = 0;
        for(i++; i > 0; i -= i&-i) s += t[i];
        return s;
    }
    void update(int i, int v) {
        for(i++; i < t.size(); i += i&-i) t[i] += v;
    }
};

int main() {
    int n; cin >> n;
    vector<vector<int>> sweep;
    for(int i = 0; i < n; i++) {
        int a, e, d; cin >> a >> e >> d;
        sweep.push_back({e, a, d});
    }
    sort(sweep.begin(), sweep.end());
    Fenwick BIT(100001);
    long long ans = 0, lst = -1;
    vector<int> upd;
    for(auto &e : sweep) {
        if(e[0] != lst) {
```

```
            for(auto &v : upd) BIT.update(v, 1);
            lst = e[0];
            upd.clear();
        }
        upd.push_back(e[1]);
        ans += BIT.value(e[1]-1)-BIT.value(max(0, e[1]-e[2]-1));
    }
    cout << ans << endl;
}
```

6 Omkar and Leaders

One way to solve this is with square-root decomposition. For every chunk, we maintain a suffix array that stores the answer to a query for the last i soldiers, which we precalculate for every chunk, and then recalculate for the corresponding chunk in every update query.

Let $dp[i]$ be the total skill from soldier i to the end of the chunk, $sum[l:r]$ be the total skill level of soldiers l to r (without leaders), and soldier j be the first soldier after solder i such that $s_j \geq s_i$. Clearly, the leader of every soldier before soldier j is soldier i, so $dp[i] = sum[i:j-1] + (j-i)s_i + dp[j]$. All that remains is to efficiently find j. This can be done in $O(\log^2 N)$ time with a binary search and range maximum queries (which is supported efficiently with a segment tree).

Updating a value is trivial—simply recalculate the suffix array for the corresponding chunk, and update the sum and range-max segment trees. To answer a query, maintain a running max m of all the soldiers. Finding the answer for a chunk is very similar to finding the suffix array; determine the first soldier j such that $s_j \geq m$. Then, the chunk contributes $sum[1:j-1] + (j-1)m + dp[j]$ to the total answer. This is assuming the entire chunk is part of the query range; if not, then brute force the part of the chunk that lies within the query range.
Time Complexity: $O((N + Q\sqrt{N})\log^2 N)$

This question is very difficult, even for the last question of the contest, and was not solved by anyone during the competition. It is roughly the same difficulty as a USACO Open Platinum question.