Editorial: NOI 2022 - Gym Badges

Avighna

June 2024

This problem uses a technique known as an 'exchange argument'. This technique involves analyzing two cases of what happens when we challenge two gyms i and j: the first one being us challenging gym i first and the second being us challenging gym j first.

If we challenge gym i before gym j

Let our level before challenging either one of the gyms be L. To fight gym i, $L \leq i_L$ must be true (that is, our level must be less than the gym's level cap). Challenging this gym would gain us i_X levels, making our new level $L + i_X$. Next, we challenge gym j, necessitating $L + i_X \leq j_L$. Our level after this becomes $L + i_X + j_X$.

To be able to challenge both of these gyms, we need to satisfy two conditions simultaneously: $L \leq i_L$ and $L \leq j_L - i_X$. The stricter (for example, if we had $x \leq 2$ and $x \leq 3$, $x \leq 2$ is the stricter condition) out of these two conditions will be the one with the smaller value, so we can combine both of these into $L \leq \min(i_L, j_L - i_X)$.

If we challenge gym j before gym i

Mirroring the steps we've done above, we would obtain the condition for this being possible as $L \leq \min(j_L, i_L - j_X)$.

Let's say you had the choice between having to satisfy $L \leq 2$ or $L \leq 5$ to obtain 2 gym badges (as in, 'I will give you 2 gym batches if your current level is ≤ 5 or ≤ 2). Which one would you choose? As you might have guessed, $L \leq 5$ is the better choice here, since it works with more L values than $L \leq 2$.

Conclusion: We would prefer to challenge gym i before gym j if $\min(i_L, j_L - i_X) > \min(j_L, i_L - j_X)$ (as this would give us more leeway in our choice for L).

Let us sort the gyms using this comparator:

```
typedef long long ll;

struct Gym {
    ll L, X;
};

int main() {
    ll n;
    std::vector<Gym> a(n);
    // ... take input
    std::sort(a.begin(), a.end(), [](const Gym &i, const Gym &j) {
        return std::min(i.L, j.L - i.X) > std::min(j.L, i.L - j.X);
    });
}
```

Note that this does not mean that we will necessarily challenge only a prefix of these gyms (it could very well be possible that our optimal solution excludes challenging some gym from the middle). Rather, our optimal solution will be a subsequence of a, and we will challenge them in order from left to right.

The solution now is to greedily pick gyms. Let us maintain our current level (L) along with all gyms we have picked so far (initially empty). If $L \leq L_i$, then we can challenge the i^{th} gym. Otherwise, keeping in mind that each gym gives us exactly 1 gym batch, it would perhaps be worth it to remove no more than 1 gym from our list of challenged gyms if the gym we're looking at increases our levels by less than the gym we're removing.

Which gym would we remove? If $L > L_i$, then removing the gym with the largest i_X value would give us the highest chance of satisfying $L \leq L_i$ after the removal. Notice that this also gives us the highest chance of increasing our levels more than the gym that we're potentially adding. So we do exactly that.

Note that this works only because we have ordered our gyms in such a way that we will always challenge them in the most optimal order. At the end, output the number of gyms that you have challenged, and you're done!

The code for this is as follows. Note that I have used an `std::multiset` to store the gyms, but using an `std::priority_queue` would work too.

```
#include <bits/stdc++.h>

typedef long long ll;

struct Gym {
    ll L, X;
};
bool operator<(const Gym &a, const Gym &b) {
    if (a.X != b.X) {
        return a.X < b.X;
    }
    return a.L < b.L;
}

int main() {
    ll n;
    std::cin >> n;
    std::vector<Gym> a(n);
    for (ll i = 0; i < n; ++ i) {
        std::cin >> a[i].X;
    }
    for (ll i = 0; i < n; ++ i) {
        std::cin >> a[i].L;
    }
    std::sort(a.begin(), a.end(), [](const Gym &i, const Gym &j) {
        return std::min(i.L, j.L - i.X) > std::min(j.L, i.L - j.X);
    });
    std::multiset<Gym> dp;
    ll level = 0;
    for (ll i = 0; i < n; ++ i) {
        if (dp.empty() || level <= a[i].L) {
            dp.insert(a[i]);
            level += a[i].X;
        } else if (level - dp.rbegin()->X <= a[i].L && dp.rbegin()->X > a[i].X) {
            level -= dp.rbegin()->X;
            dp.erase(--dp.end());
            dp.insert(a[i]);
            level += a[i].X;
        }
    }
    std::cout << dp.size() << "\n";
}
```