USACO 2023 Bronze Problems

USACO doesn't let us access the problems again until the results are being processed or something so bear with me through this :(

Problem 1 (Candy Canes)**.** This one is sort of implementation.

Right away from the problem description, we see that after each round of cows, the candy cane is thrown away, so we can safely just loop through all the candy canes and then go through the rounds of cows. We'll have a variable keeping track of the height of the candy cane and increment it accordingly based on however much a cow eats.

Notice that the amount the cow eats is the minimum between the length of the cane and the difference between the height of the cow and the height of the cane (in other words, the cow can only eat as much as there is in the cane, and it will eat as tall as it can go).

We can then add this eaten amount to the height of the cane, the height of the cow, and we will subtract it from the length of the cane.

One remark is, if you simply iterate through the cows (replacing the inner while loop with a for loop), you run out of time, so make sure to keep early exit in mind.

```
#include <iostream>
#include <algorithm>
#include <vector>

#define ll long long

int main() {
    /* This is just for reading input */
    std::vector<long long> cows;
    std::vector<long long> canes;

    long long N;
    long long M;

    std::cin >> N;
```

```
std::cin >> M;

for (ll i = 0; i < N; i++) {
    ll a;
    std::cin >> a;
    cows.push_back(a);
}

for (ll i = 0; i < M; i++) {
    ll a;
    std::cin >> a;
    canes.push_back(a);
}
/* End of reading input */

/* The actual algorithm */
for (ll i = 0; i < M; i++) {
    ll height = 0;
    ll cane = canes[i];

    ll j = 0;
    while (cane > 0 && j < N) {
        ll cow = cows[j];
        if (cow > height && cane > 0) {
            ll eaten = std::min(cane, cow - height);
            height += eaten;
            cows[j] += eaten;
            cane -= eaten;
        }

        j++;
    }
}

/* Printing the solution out */
for (ll i = 0; i < N; i++) {
    std::cout << cows[i] << std::endl;
}
```

```
}
```

Perhaps there's a smarter implementation of this, but it gets the job done so yay.

Problem 2 (Cowntact Tracing)**.** Frick problem 2 all my homies hate problem 2.

Okay so this is actually just going to be a half solution because I only got half of the test cases to work, and I don't exactly know why (hopefully Joseph tells me where I'm being dumb). Also, my solution method and code are actually garbage, so whoo boy buckle up.

A natural simplification to the problem is to group up the blocks of 1's and instead store their length. In addition, we'll consider blocks in the middle and blocks at the ends to be separate, because they sort of behave differently.

There's probably a way to do this without considering the number of days, but I tried doing it this way anyway. Let's consider the middle blocks first, which we'll denote the lengths of (if existing) as $m_1, m_2, \ldots, m_n$. No matter what, these will grow by 2 each day, so the maximum number of days they could have grown for is $\lfloor (m_i - 1)/2 \rfloor$ (if its an even length it will have started out at 2, and if its an odd length it will have started out at 1). If we take the minimum of all of these middle maximum possible days, we get the maximum possible number of days that the entire middle section could have grown for:

$$d_{\text{mids}} = \min_i \left\lfloor \frac{m_i - 1}{2} \right\rfloor.$$

Now let's consider the ends, which are slightly different. Let us denote the length of the ends (if existing) by e_1, e_2. The maximum number of days they could have grown for is one less than their length (this case is achieved by putting the 1 at the very end at the start):

$$d_{\text{ends}} = \min\{e_1 - 1, e_2 - 1\}.$$

Taking the minimum of the two, we achieve the maximum feasible number of days that the virus could have grown for.

$$d = \min\{d_{\text{mids}}, d_{\text{ends}}\}.$$

It makes intuitive sense that, by maximizing the number of days, we minimize the number of cows (actually I also tested out trying every time up to it as well and taking the minimum cows, but that also didn't work so I am very sad).

For the middle blocks, the number of cows that existed at the start before the d days is going to be $m_i - 2d$ (because the middle ones grow by 2 everyday).

For the end blocks, one can work out that the number of cows that need to be placed is[1] $\lceil e_i/(2d+1) \rceil$.

Combining these, our final answer should be

$$S = \lceil e_1/(2d+1) \rceil + \lceil e_2/(2d+1) \rceil + \sum_i (m_i - 2d).$$

But yeah idk. Also yeah don't entirely trust the names I give to my variables when coding.

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <cmath>

int main() {
    int N;
    std::cin >> N;

    std::vector<int> mids;
    std::vector<int> ends;

    bool end = false;
    int current = 0;

    for (int i = 0; i < N; i++) {
        char c;
        std::cin >> c;

        if (c == '1') {
            current++;
```

[1] Probably. Then again I could have also gotten this very wrong because half of the test cases were wrong :(

```
        if (i == 0) {
            end = true;
        }
    } else if (current > 0) {
        if (end) {
            ends.push_back(current);
        } else {
            mids.push_back(current);
        }
        end = false;
        current = 0;
    }
}

if (current > 0) {
    ends.push_back(current);
}

std::vector<int> middle_times(N);
int t_max = std::numeric_limits<int>::max();

for (int i = 0; i < mids.size(); i++) {
    middle_times[i] = (mids[i] - 1) / 2;
    t_max = std::min(middle_times[i], t_max);
}

if (ends.size() == 0) {
    int S = 0;

    for (int i = 0; i < mids.size(); i++) {
        S += mids[i] - 2 * t_max;
    }

    std::cout << S << std::endl;
} else {
    int ends_max = *std::min_element(ends.begin(), ends.end()) - 1;

    if (mids.size() == 0) {
```

```
            int S = 0;

            for (int i = 0; i < ends.size(); i++) {
                S += (int)ceil((double) (ends[i]) / (2 * ends_max + 1));
            }

            std::cout << S << std::endl;

            return 0;
        } else {
            int best_time = std::min(ends_max, t_max);
            std::vector<int> sums;

            for (int t = 0; t <= best_time; t++) {
                int S = 0;

                for (int i = 0; i < mids.size(); i++) {
                    S += mids[i] - 2 * t;
                }

                for (int i = 0; i < ends.size(); i++) {
                    S += (int) ceil((double) ends[i] / (2 * t + 1));
                }

                sums.push_back(S);
            }

            int smallest = *std::min_element(sums.begin(), sums.end());

            std::cout << smallest << std::endl;
        }
    }

    return 0;
}
```

Problem 3 (Farmer John Actually Farms)**.** Actually this problem was a pretty fun one. I also remember it more so I'll give a rough description of the problem. We have N being the number of plants, $h = [h_0, h_1, \ldots, h_{N-1}]$ being the heights of each respective plant,

$a = [a_0, a_1, \ldots, a_{N-1}]$ being the growth rates of each plant, and $t = [t_0, t_1, \ldots, t_{N-1}]$ being the number of plants that Farmer John wants to be taller than the respective plant (basically the sorting of the plants).

Each plant grows by its growth rate every day and we want to find out if the configuration of tallest to shortest that Farmer John asks us for is possible or not and, if so, return the minimum number of days for it to be achieved.

The first thing that we must do to make our lives a ton easier is construct an index mapping that lets us go from the sorted indices to the unsorted indices of plants that are given to us. In other words, let I be our index map and $I[t[i]] = i$.

The algorithm that we'll use will go from ordered indices $j = 1, 2, \ldots, N-1$ and make sure that the plant at ordered index $j-1$ (really at index $I[j-1]$, but that's what I mean by ordered index) can be greater than the plant at j after some number of days. By virtue of comparing the previous plant to the current plant and incrementing the index, we make sure that all plants with a greater index will be shorter.

The key insight is to realize that, in order to verify the condition that the previous plant can outgrow the other, we have a very interesting linear inequality. Let $h_1 = h[I[j-1]], a_1 = a[I[j-1]]$ and $h_2 = h[I[j]], a_2 = a[I[j]]$ (that is, h_1 and a_1 represent the height and growth rate of the previous plant which should be taller and the opposite holds for h_2 and a_2). We have that

$$h_1 + a_1 d > h_2 + a_2 d \iff d\,(a_1 - a_2) > h_2 - h_1.$$

For certain cases of heights and growth rates, the inequality flips and we get a upper bound for the number of days. For some cases, it doesn't and we get an lower bound. For other cases, we get no information. For some cases, we get that the goal is impossible and we can exit out and print -1. In this way, we can keep track of running lower and upper bounds for the number of days it takes and at the end (if we get that it is a possible configuration) we shall output the lower bound, which is the minimum number of days it will take. It is now our job to go casewise and do some thinking.

1. Case $a_2 \geq a_1$ and $h_2 \geq h_1$. In this case, the previous plant can never outgrow the current one, so we can exit out saying that the configuration is impossible.

2. Case $a_2 > a_1$ and $h_2 < h_1$. In this case, the inequality flips and we get an upper

bound for the number of days. We have that

$$d < \frac{h_2 - h_1}{a_1 - a_2},$$

but because of the strict inequality, we get that the upper bound is

$$d_{\text{upper}} = \left\lceil \frac{h_2 - h_1}{a_1 - a_2} \right\rceil - 1.$$

We take the minimum of the current running upper bound and this and update the running upper bound.

3. Case $a_2 < a_1$ and $h_2 \geq h_1$. In this case, the inequality doesn't flip and we get a lower bound for the number of days. We have that

$$d > \frac{h_2 - h_1}{a_1 - a_2},$$

and we take the strict inequality into account to get that the lower bound is

$$d_{\text{lower}} = \left\lfloor \frac{h_2 - h_1}{a_1 - a_2} \right\rfloor + 1.$$

We take the maximum of this with the running lower bound and update the running lower bound.

4. Otherwise, we gain no information

After every change to the upper or lower bounds, we should also check that the upper bound is not less than the lower bound. If it is, we can exit early and say that the configuration is impossible.

As previously stated, if all goes well and there are no contradictions, we may return the running lower bound value that achieves the minimum number of days.

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
```

```
#define ll long long

ll solve(ll N, std::vector<ll> &h, std::vector<ll> &a, std::vector<ll> &t) {
    std::vector<ll> index_map(N);

    for (ll i = 0; i < N; i++) {
        index_map[t[i]] = i;
    }

    ll d_lower = 0;
    ll d_upper = std::numeric_limits<ll>::max();

    if (N == 1) {
        if (t[0] == 0) {
            return 0;
        } else {
            return -1;
        }
    }

    for (ll j = 1; j < N; j++) {
        ll i = index_map[j];
        ll prev = index_map[j - 1];

        ll h_prev = h[prev];
        ll a_prev = a[prev];

        ll h_i = h[i];
        ll a_i = a[i];

        if (a_i >= a_prev && h_i >= h_prev) {
            return -1;
        } else if (a_i == a_prev && h_i < h_prev) {
            // Perhaps redundant because we have an else continue,
            // but this made more sense in my head while trying to
            // organize everything
            continue;
```

```
        } else if (a_i > a_prev && h_i < h_prev) {
            ll A = (h_prev - h_i) / (a_i - a_prev);
            if ((h_prev - h_i) % (a_i - a_prev) == 0) {
                A--;
            }
            d_upper = std::min(A, d_upper);
        } else if (a_i < a_prev && h_i >= h_prev) {
         d_lower = std::max(1 + (h_prev - h_i) / (a_i - a_prev), d_lower);
        } else {
            continue;
        }

        if (d_upper < d_lower) {
            return -1;
        }
    }

    return d_lower;
}

int main() {
    ll T;

    std::cin >> T;

    for (ll i = 0; i < T; i++) {
        ll N;
        std::cin >> N;

        std::vector<ll> h;
        std::vector<ll> a;
        std::vector<ll> t;

        for (ll j = 0; j < N; j++) {
            ll n;
            std::cin >> n;
            h.push_back(n);
        }
```

```
        for (ll j = 0; j < N; j++) {
            ll n;
            std::cin >> n;
            a.push_back(n);
        }

        for (ll j = 0; j < N; j++) {
            ll n;
            std::cin >> n;
            t.push_back(n);
        }

        std::cout << solve(N, h, a, t) << std::endl;
    }

    return 0;
}
```

Okay I am done now excuse all my missed punctuation and perhaps typos