USACO

There are many programming competitions available for high school students in the United States. Most are local contests hosted by various universities or companies. (For example, in the Central Florida area, Lockheed-Martin, University of Central Florida, Stetson and University of Florida have all hosted programming competitions for high school students.)

One competition that occurs nationwide in the United States (and is specifically meant for high school students) is the United States of America Computing Olympiad. It is most analogous to the AMC (American Mathematics Competition), AIME (American Invitational Mathematics Exam) and USAMO (United States of America Mathematics Olympiad) sequence of competitions for mathematics. In mathematics, many students (about 160,000 in 2023) took the AMC nationwide. Based on those scores, roughly 3,000 students qualify to take the AIME. Based on a composite of AMC and AIME scores, about 300 students are chosen to take the USAMO. Top students from the USAMO are invited to a camp and then students from that camp are chosen to represent the United States at the International Mathematics Olympiad (IMO).

Instead of having separate tiers and competition formats as mathematics, the United States relies on USACO to ultimately choose the United States team for the International Olympiad of Informatics (IOI). Here is how USACO is structured differently than the process mathematics uses to select IMO candidates:

1. Competitions are given four times a year, typically in December, January, February and March of an academic year.

2. For each competition, students can compete from anywhere (not necessarily being proctored at a set time at a school) and have a four day window to compete for each competition.

3. Each competition has four different levels: Bronze, Silver, Gold and Platinum.

4. All students, if they've never competed, start out in Bronze. To get into any other level, a student must compete and score well enough (typically either over 750 or 800 out of 1000) to be promoted to the next level. Once a student gets promoted to the next level, they can not be "depromoted." (Surprisingly, this is both good and bad…)

5. Each competition has 3 competitive programming style questions and students have four hours (or five hours for the last competition) to submit programs to solve the problems.

6. Partial credit is awarded for each question. For each submission to a question, it is tested against several test cases. Each test case is worth the same amount of points (within 1), and the sum of the points for all the test cases for a single problem is 333 (or 334). Students receive the score associated with their last submission to a problem and can submit as many times as they want on a question after getting feedback on each test case (Correct, Wrong Answer, Time Limit Exceeded, Run Time Error).

Roughly speaking, here is a quick description of the level of questions at each level:

Bronze: Minimally tests complicated algorithms taught in a formal setting. Instead, most of the questions require students making observations and using standard language tools (sorting, built in data structures like sets or maps). Often times, there are problems here that young students good at general problem solving can get while those who've formally learned Computer Science might miss. The question writers try fairly hard to make sure that someone who hasn't been taught something, but is good at making observations and reductions, has a shot at solving these problems.

Silver: These questions tend to test more formal algorithms and data structures taught in typical computer science courses, but compared to a homework question, many more observations need to be made, there's more problem obfuscation and there's more emphasis on efficient algorithms as compared to Bronze. Algorithms like breadth first search, minimum spanning tree and the like often show up at this level.

Gold: These questions start getting into more advanced algorithmic topics such as dynamic programming and trees. In addition, often times, these tend to require more mathematical knowledge than typical collegiate programming contests. (Likely this is due to the fact that most of the top former participants who make questions often are also past USAMO participants.) In addition, the problem obfuscation tends to be at a higher level than silver.

Platinum: These questions are really quite hard. A vast majority of college competitors can't fully solve most of these questions. They involve all sorts of complicated techniques that are generally only taught to competitive programmers and are beyond the standard undergraduate curriculum. Also, many of these tend to be even more mathy than the Gold questions.

Just like the AMC/AIME/USAMO set of competitions, doing well in USACO can be helpful when applying to colleges, especially if you're interested in a technical degree of some sort. So, if you enjoy programming competition problems, I strongly suggest USACO participation because it's much easier than most competitions (you can compete from home in a wide time window) and it still has the benefit of being recognized as a very difficult academic challenge.

The Website: usaco.org

First, go to the website usaco.org and create a login. After you do that, there are two ways to practice contest problems:

1. Register for a real contest and compete.
2. Solve individual problems from an old contest.

Submitting Problems on USACO

In the 2020-2021 season, USACO changed to use standard input/standard output. Before that, USACO used file input/file output. This means that if you are running a live contest in 2021 or later, please use standard input and output. If you are practicing on an old problem prior to the 2020-2021 school year, please use file input and file output. The directions for using files is on the next page.

Old File Input/Output Directions

For each individual problem, a filename is specified. The input file you read from must be the filename.in and the output you produce must go to the file filename.out. For example, if a problem has the filename "greetings,". The solution must read from "greetings.in" and write output to "greetings.out." In C++, here is how to set up a stream to read from an input file.

```
ifstream fin("greetings.in");
```

Now, to read, use just like you use cin, but just write fin:

```
fin >> n;
```

After finishing reading from a file, close the file:

```
fin.close();
```

To write to a file, open it as follows:

```
ofstream fout("greetings.out");
```

Write to the file as follow:

```
fout << n << endl;
```

and close the file when done:

```
fout.close();
```

Later in these notes, a solution to the December 2016 Bronze Problem: Square Pasture (filename square) is included, illustrating the use of the syntax above. When you are on the page of a USACO problem, there's a pull down menu at the bottom of the problem description for the language of your solution and a second button where you click to upload your source file. After you do that, just it the "Submit Solution" button. When you do, you'll see your results on each test case at the top of the problem description window. A test case your program solves correctly will be a green box with a '*' character and the amount of memory and time your program took on that case. All incorrect test cases are indicated with a red box, and the character in the box indicates whether the response is a Wrong Answer ('x'), Time Limit Exceeded ('t'), or Run Time Error ('!').

Two ways to Practice:

1. Live Contests on USACO

On the main USACO page there's a schedule on the top right. That shows four contests and the window of days for the contest. Each contest has a 4 day window. To compete in a contest, you must login during that four day window and then hit a button at the top of the page that says, "Compete in December 2024 Contest" (or something of that nature). Once you click on that, there will be another page that comes up with the contest rules, and if you scroll to the bottom, you can just hit the button "Start my 4 hour contest window now" (or something to that effect). Once you do that, three problems will pop up in your main browser window and you can click on any problem and read it. During contest, you can see your results as previously described. USACO does partial credit, so if you get some test cases green, you do get some points, so, you may either choose to move on, or edit your program to see if you get more test cases correct. Your score will be based on your last submission to each problem.

2. Solving Individual Problems from Old Contests

On the main USACO page, click on the menu for Contests. This brings up a list in reverse chronological order. Click on any of the contest links you like. When you do this, you'll see a summary of the contest and if you scroll down enough, you'll see the problems at each level, Platinum, Gold, Silver and Bronze. (If you look at a contest at or before the 2015 US Open, there will be no Platinum level.) For practice, I recommend starting with Bronze problems until you feel they are too easy. Basically, you just click on a problem on any of these contests pages, and you can submit like previously discussed and see your results. (Remember, standard input/output for the 2020-2021 academic year and later, file input for 2019-2020 and before.)

For the rest of these notes, we'll look at five past USACO problems and sample solutions for each. The first will be for a problem before 2020 to illustrate the use of files for I/O. The rest will be from 2023 to illustrate more recent trends in the Bronze problems.

December 2016 Bronze Problem: Square Pasture

Here is a link to the problem:

https://usaco.org/index.php?page=viewproblem2&cpid=663

The input to the problem consists of the x-y coordinates of bottom left and top right corners of two rectangles. The goal of the problem is to find the minimum area of any axis-aligned square that completely covers both rectangles.

The key observation here is that since the square has to be axis aligned, the side length has to be at least as big as the difference between any pair of x coordinates, and it also has to at least as big as the difference between any pair of y coordinates.

So one possible solution is as follows: maintain the minimum x, maximum x, minimum y and maximum y values as the data values are read in. Then, calculate the two differences, maximum x minus minimum x and maximum y minus minimum y. Of these two differences, calculate the maximum value. The answer to the question is just the square of this number.

Here is the code, including using files for input and output named “square.in” and “square.out”:

```
// Arup Guha, 6/18/2024
// Solution to the 2016 Dec USACO Bronze Problem: Square Pasture
using namespace std;
#include <bits/stdc++.h>

int main() {

    // Open input file.
    ifstream fin("square.in");
    int x, y;
    fin >> x >> y;

    // Set these based on first corner.
    int minx=x, miny=y, maxx=x, maxy=y;
    for (int i=0; i<3; i++) {
        fin >> x >> y;
        minx = min(minx, x);
        maxx = max(maxx, x);
        miny = min(miny, y);
        maxy = max(maxy, y);
    }

    // Close file.
    fin.close();

    int side = max(maxx-minx, maxy-miny);

    ofstream fout("square.out");
    fout << side*side << endl;
    fout.close();
    return 0;
}
```

January 2023 Bronze Problem: Air Cownditioning

Here is a link to the problem:

https://usaco.org/index.php?page=viewproblem2&cpid=1276

In this problem, there are several cows, each of which live in some set of contiguous barns. (Barns are numbered 1 to 100.) Each cow has their own space, so the sets of contiguous barns for each cow do not intersect at all. Each cow needs to be cooled. Specifically each one has a requirement that they must be cooled by some number of degrees, meaning that each one of their barns must be cooled by at least that many degrees. There are upto 10 airconditioners that Farmer John can buy. Each one can cool some contiguous range of barns by some number of degrees and has some cost. The goal is to figure the minimum cost to keep all the cows happy.

Since you are buy any subset of airconditioners, and there are only 10, this is very clearly testing brute force. The solution is just to try all subsets of airconditioners (so $2^{10} = 1024$ subsets), and for each subset, calculate its cost and simulate its cooling. (The run time for simulating cooling would be no more than 10 x 100 = 1000 simple steps. At most there are 10 airconditioners, and each cools at most 100 barns.) Once we are done, we can simply just look through each cell to see if it was cooled enough. If it was, we see if the cost for this subset is less than the best previous answer. If so, update our answer.

In the solution below, bitmasks are used to loop through each subset. In addition, arrays are used instead of vectors, but the syntax, except for declaration, is basically identical to vector syntax.

The go function takes in a mask representing the subset of AC units that are bought, and then simulates each of these cooling the barns (just by looping rhough and adding to each barn's cooling value), and then checks if each barn is cool enough. If so, the function returns the corresponding cost to buy those ACs. If not, -1 is returned to indicate that that subset doesn't fulfill the cooling requirement. Here's the code:

```
// Arup Guha
// 2/4/2023
// Solution to USACO Jan 23 Problem: Air Cownditioning II

using namespace std;

#include <iostream>

typedef struct ac {
    int sI; // start range
    int eI; // end range
    int c; // cooling
    int money; // cost...
} ac;

// To give access to my function(s).
ac acList[10];
int need[100];
int numCow, numAC;
int go(int mask);
```

```
int main() {

    // Read in main parameters.
    cin >> numCow >> numAC;

    // This is how much cooling we need.
    for (int i=0; i<100; i++) need[i] = 0;

    // Read in cows and update cooling requirement.
    for (int i=0; i<numCow; i++) {

        // Read in info, make 0 based indexes.
        int s, e, c;
        cin >> s >> e >> c;
        s--; e--;

        // Update the need in these cells.
        for (int j=s; j<=e; j++) need[j] += c;
    }

    // Read in AC info. Set init info to all ACs being on.
    int res = 0;
    for (int i=0; i<numAC; i++) {
        cin >> acList[i].sI >> acList[i].eI >> acList[i].c >>
acList[i].money;
        acList[i].sI--;
        acList[i].eI--;
        res += acList[i].money;
    }

    // Try each subset of ac's on.
    for (int i=0; i<(1<<numAC); i++) {

        // Evaluate turning on this set of ACs.
        int thisCost = go(i);

        // Not valid.
        if (thisCost == -1) continue;

        // Update.
        if (thisCost < res) res = thisCost;
    }

    // Ta da!
    cout << res << endl;

    return 0;
}
```

```
// Returns -1 if the subset of AC specified by mask can't meet the
// requirement. Returns the cost of turning on these ac's if it does.
int go(int mask) {
    // Will store how much these AC's cool.
    int cool[100];
    for (int i=0; i<100; i++) cool[i] = 0;

    // Cost of this subset.
    int cost = 0;

    // Go through the AC's
    for (int i=0; i<numAC; i++) {

        // We're not supposed to turn this AC on.
        if ( (mask & (1<<i)) == 0) continue;

        // Add in cost.
        cost += acList[i].money;

        // Update cooling.
        for (int j=acList[i].sI; j<=acList[i].eI; j++)
            cool[j] += acList[i].c;
    }

    // See if any cow in any stall is unhappy.
    for (int i=0; i<100; i++)
        if (cool[i] < need[i])
            return -1;

    // This is how much it cost.
    return cost;
}
```

February 2023 Bronze Problem: Hungry Cow

Here is a link to the problem description:

https://usaco.org/index.php?page=viewproblem2&cpid=1299

In the problem, shipments of food for Bessie arrive at various days. Each shipment has food for some number of days for her and the food never goes bad (only in competitive programming is this the case…) The goal is to figure out the maximum number of days, from day 1 to day T, that Bessie can eat.

We can frame the problem as follows: When food arrives at day d, if the shipment has k days worth of food, we are guaranteed to be able to eat every day from day d to day d + k – 1, for k straight days. Let's say that before day d + k – 1, a new shipment with m days of food comes. We can eat that shipment AFTER day d + k – 1, so then we're covered from day d + k to day d + k + m – 1. In code, we can keep a "current day" that we are fed until, as well as a total number of days we've eaten. Then, when a new shipment arrives, we can adjust both of these numbers in O(1) time. The one tricky case is that if day T comes before the end of the food supply. In that case, you have to note that you can eat until day T and then stop (via break statement). The solution is on the next page:

```
// Arup Guha
// 3/4/2023
// Solution to USACO Feb 2023 Bronze Problem: Hungry Cow

using namespace std;

#include <bits/stdc++.h>

typedef long long ll;

int main() {

    int n;
    ll max;
    cin >> n >> max;

    // Initially it's day 0 and we haven't eaten.
    ll curD = 0, curEat = 0;

    // Go through haybales.
    for (int i=0; i<n; i++) {

        // Read in this shipment.
        ll day, bales;
        cin >> day >> bales;

        // Update the current day if necessary.
        if (curD < day) curD = day;

        // We're done.
        if (curD > max) break;

        // Last time we eat before deadline.
        if (curD + bales - 1 > max) {
            curEat += (max-curD+1);
            break;
        }

        // We eat all of these haybales.
        else {
            curEat += bales;
            curD += bales;
        }
    }

    // Ta da!
    cout << curEat << endl;

    return 0;
}
```

February 2023 Bronze Problem: Watching Mooloo

Here is a link to the problem description:

https://usaco.org/index.php?page=viewproblem2&cpid=1301

In this problem, Bessie wants to watch Mooloo on some days. (Days are numbered.) The amount she pays for a single subscription is K + d, where d is the number of consecutive days of the subscription. (Basically, K is the start up cost, and then it's 1 per day.) Thus, it might be advantageous to get multiple shorter subscriptions, particularly if the gap between days she uses Mooloo is large.

Essentially, what we find out is that if there is a gap that is K+1 days or more, we should just get a new subscription. So, basically, the first day we watch, our cost is K+1 because we bought our subscription and have watched for one day. If the next day we watch Mooloo is K or fewer days away, keep the same subscription and just add to the cost the number of days we need to get to the new day (new day minus old day). But, if we wait longer to watch again, we may as well cut our subscription, and then get a new one for cost K and then pay 1 for our new day watching. The code turns out to be very clean:

```
// Arup Guha, 6/19/2024
// Solution to USACO Feb 2023 Bronze Problem: Mooloo
using namespace std;
#include <bits/stdc++.h>
typedef long long ll;

int main() {
    int n;
    ll k;
    cin >> n >> k;

    // Read first day, assign cost.
    ll cost = k+1;
    ll day;
    cin >> day;

    for (int i=1; i<n; i++) {
        ll tmp;
        cin >> tmp;

        // Just keep the old subscription.
        if (tmp-day <= k)
            cost += (tmp-day);

        // Just start a new one.
        else
            cost += (k+1);

        // Update day.
        day = tmp;
    }
    cout << cost << endl;
    return 0;
}
```

December 2023 Bronze Problem: Candy Cane
Here is a link to the problem description:

https://usaco.org/index.php?page=viewproblem2&cpid=1347

In this problem, we have many candy canes, numbered 1 through M, hanging from various heights so that they just touch the ground. There are many cows numbered 1 through N, who each get a chance to eat the candy canes. The process is as follows: Candy Cane #1 is presented to the cows to eat. Cow #1 goes and eats as much as she can. If she's height 10 and the candy cane is height 17, she eats 10 units of candy cane and now the candy cane is hanging from height = 10 to height 17. Next, Cow #2 goes and east as much as she can. If she's height 13, she'll get to eat 3 units of candy cane, from height = 10 to height = 13, leaving 4 units of candy cane left from height = 13 to height – 17. Next, Cow #3 goes and eats as much as she can. If she's height 12, then she gets no candy cane, because 13 > 12. The process continues until all cows get a chance. **The interesting thing is that after the round of eating, each cow grows by how much candy cane she ate.** Then the process is repeated for Candy Cane #2, then Candy Cane #3 and so forth. The goal of the problem is to calculate how much candy cane each cow eats.

With upto 200,000 cows and 200,000 candy canes, there isn't time to simulate the process as described. However, let's think about reordering the process so that the same cows get the same amount of candy cane:

Let Cow #1 eat all she can before we consider any other cows. This is possible because she always goes first, so we know where the candy canes will be. We just have to simulate her getting taller between candy canes. (She basically gets the minimum of her height and the candy cane height for each candy cane.) Consider the case where cow 1 is short, say height 1. After eating the first candy cane, she's height 2. After eating the second candy cane, she's height 4 (or the second candy cane is gone…) After the third one, height 8. Basically, every time a candy cane DOESN'T disappear, she doubles in height, at least. Since the maximum height candy cane is 10^9, she can double in height at most 29 times. This means that the vast majority of the candy canes will disappear after the first round, no matter what. (Something like 29 will be left at most…)

This means that for the rest of the cows numbered 2 through 200,000, we have enough time to simulate our process we did with cow #1, since 200,000 x 30 = 6,000,000 only, which is few enough operations to run in time.

In the code starting on the next page, we start with a list of original candy canes, and then after each cow, create a new list of candy canes that remain after the previous cow ate. So, the outer loop is through the cows and the inner one through the candy canes. As described above, we know that only the first time is the candy cane list big. For all subsequent iterations, the list size is no more than 29. Here's the code:

```
// Arup Guha
// 6/19/2024
// Solution to 2023 December USACO Bronze Problem: Candy Cane Feast
using namespace std;
#include <bits/stdc++.h>
typedef long long ll;

int main() {

    int nCows, nCanes;
    cin >> nCows >> nCanes;
    vector<ll> cows(nCows);
    for (int i=0; i<nCows; i++)
        cin >> cows[i];

    // First item is how tall, second is how far off the ground it hangs.
    vector<pair<ll,ll>> canes(nCanes);
    for (int i=0; i<nCanes; i++) {
        cin >> canes[i].first;
        canes[i].second = 0;
    }

    // Go through cows.
    for (int i=0; i<nCows; i++) {

        // We'll put candy canes that make it to the next round here.
        vector<pair<ll,ll>> newCanes;

        // Loop through the candy canes.
        for (int j=0; j<canes.size(); j++) {

            // Can't eat anything this goes to the next round.
            if (cows[i] < canes[j].second)
                newCanes.push_back(canes[j]);

            // We eat sum.
            else {

                // This is what we eat.
                ll   eat   =   min(canes[j].first-canes[j].second,   cows[i]-
canes[j].second);

                if (cows[i] < canes[j].first)
                    newCanes.push_back(pair<ll,ll>(canes[j].first,cows[i]));

                // Update my height.
                cows[i] += eat;
            }
        }

        // Update our list of candy canes!
        canes = newCanes;
    }
    for (int i=0; i<nCows; i++)
        cout << cows[i] << endl;
    return 0;
}
```