Problem 1: Ice Cream Bars!
5+4+4=13 Points

Problem ID: `bars`
Rank: `1+2+3`

Introduction

Problem Summary

Given $\mathbf{N}$ ice cream bars, find the maximum number of days you can follow this diet: On the first day you eat 1 bar, on the second day you eat 2, on the third day you eat 3, and so on.

You can find the original problem statement here:
https://calico.berkeley.edu/files/calsuco-22/contest/bars/bars.pdf

Big Ideas

For our first CALSUCO problem, we wanted something straightforward to get contestants of all levels warmed up. While the main test set can be solved through naive simulation, the first bonus required a little math, and the second bonus required an optimized computation.

We saw a lot of contestants attempt and miss the bonuses after getting the main test set (they had a combined 16% success rate!) but we want to remind everyone that the bonuses are optional—you don't need them to continue on to other problems!

We also wanted to mention that the second bonus for this problem is quite different from the problems we typically feature, as we wanted to try some unique problems out in this contest. It highly favors Python and Java programmers while C++ programmers are disadvantaged. Furthermore, it was more knowledge-based than problem solving. Our rationale behind this is that we think using the right language for the right job along with the ability to use a search engine quickly are important skills for any programmer to master. That being said, we recognize that not everyone was a fan of this test set and you can expect test sets like these to be rare in future contests. This is also the reason why we made it worth the least points out of all the problems.

Main Test Set

This test set can simply be solved by simulating the scenario described.

We Live in a Simulation *Solutions available in C++, Java, Python*

We can begin by creating an integer variable `day` to track the current day starting at 0, and another integer variable `bars` to track the number of bars we have, starting at $\mathbf{N}$. Then, we can use a while loop to go through the actions of each day. The while loop checks if it's possible to eat the desired amount for the next day, and if so, update our variables accordingly. To do this, we check to see if `bars - (day + 1) >= 0` is true. If it's possible, we increment `day` and subtract `day` from `bars`. When the while loop is no longer able to run, that means we can't eat for any more days, so we output `day` and we're done.

To analyze the time and space complexity of algorithms, computer scientists use something called big O notation. If this is your first time hearing about this, we highly recommend you check out this article to learn the basics.

Since each individual step of the algorithm performs a $O(1)$ operation on our integer variables, it shouldn't be too hard to see that the time complexity is proportional to the number of while loop iterations run. Fortunately, a very well known math formula comes to the rescue! We'll discuss it more in the next section, but if we take the inverse, we find that the number of loop iterations is $O(\sqrt{N})$, so our overall time complexity is $O(\sqrt{N})$ as well.

Bonus Test Set 1

This test set is large enough that a naive simulation approach exceeds the time limit, and 32 bit integers aren't large enough to contain the value of $\mathbf{N}$. However, we can use some math, different data types, and new techniques to find an answer more efficiently.

There's a well known formula that gives the sum of the first d natural numbers:

$$\sum_{i=1}^{d} i = 1 + 2 + \ldots + d = \frac{d(d+1)}{2}$$

Legend has it that Gauss discovered it himself in elementary school!

This formula is useful because we can interpret the summands ($1 + 2 + \ldots + d$) to be the bars we eat each day, the final sum to be the bars we've eaten in total, and n to be the days we've eaten for. However, the problem with this formula in our case is that it tells us not how many days we can eat given N bars, but rather how many bars are needed to fully eat for d days.

Long Binary Search *Solutions available in C++, Java, Python*

One way to get around this issue is to observe that if we have enough bars for d days, then we must also have enough bars for d - 1 days too! Our formula efficiently checks if it's possible to fully eat for d days given N bars. From here, finding the answer can be done by running a binary search over the range from 0 to $\mathbf{N}$ to search for the largest number of days d that we can eat for. If the number of bars we want to eat $\frac{d(d+1)}{2}$ exceeds the number of bars we actually have $\mathbf{N}$ then we know the answer d must be in the lower interval. Otherwise, we can search for a better answer in the higher interval. Continue this while tracking the largest valid d so far, and we will eventually have our answer.

Since 32 bit integers are too small, we can use a 64 bit integer to perform our calculations instead. If you use Python, you don't need to do anything additional because `int` already handles numbers of this size. If you use Java, you can use a `long`. If you use C++, you can use a `long long int`.

The time needed for a binary search is equal to the logarithm of the size of the search space multiplied by the time needed per iteration. Since our search space is $\mathbf{N}$ and each iteration takes $O(1)$ to perform simple arithmetic, our overall runtime is $O(log(N))$.

Another way to get around this issue is to directly find the inverse of the formula! If we expand out the formula denoting the number of bars needed, we get a quadratic equation:

$$N \geq \frac{d(d+1)}{2} = \frac{1}{2}d^2 + \frac{1}{2}d + 0$$

Note that we use $\geq$ here because the number of bars we have should be at least as much as we eat to be valid. Now, we can subtract N from both sides and apply the quadratic formula from basic algebra to solve for d in terms of N. Alternatively, you can also just use a solver like Wolfram Alpha. After some simplifying, we get:

$$d \leq \frac{1}{2}(\sqrt{8N + 1} - 1)$$

Note that we can ignore the negative domain solutions because eating negative bars doesn't make sense. This formula tells us that d, the number of days we can eat, should be no more than the expression involving N, the number of bars we have, on the right. To find the largest value of d possible, we simply plug in $\mathbf{N}$ and round our answer down to the nearest integer for our final answer.

Performing this calculation using 64 bit floating point numbers is sufficient for this test set. If you use Python, you can use a `float`. If you use Java, you can use a `double`. If you use C++, you can use a `double` as well.

A fixed number of operations are performed on a fixed sized data type, so our runtime is $O(1)$.

However, there is a major caveat with this approach. Floating point isn't exact, and performing multiple calculations can lead to errors that compound to become problematic over time. Furthermore, floats can't actually represent every single integer in their range! In other words, there may be some values relevant to your calculation that simply cannot be correctly stored, causing you to inevitably get a wrong answer, even if your formula is mathematically correct.

For this reason, we intentionally chose the maximum value of $\mathbf{N}$ of 10^{15} to be smaller than the 64 bit floating safe integer limit of 2^{53} - 1 and constructed test cases that try to avoid floating point calculation errors. In general, it's not a good idea to use floating point numbers when exact integer values are required. But when an opportunity arises, it can be tremendously useful to use floats because they are fast to calculate and easy to implement. On the other hand, computing an exact integer square root is a much more computationally expensive task that we'll explore in the next section.

Bonus Test Set 2

This set test poses a unique challenge because $\mathbf{N}$ can contain values up to 10^{10000}, which is larger than any standard data type. To solve this bonus, we need to combine our formula in the previous section with a way of storing and efficiently computing extremely large integers.

Arbitrary Precision *Solutions available in Java, Python*

The main idea behind representing extremely large integers is that by chaining together many integers in an array, we can interpret the entire thing as a single, large integer. Since numbers represented in this format can be as large or small as we want, we call it arbitrary precision.

Fortunately, arbitrary precision is an issue that has been tackled by many programmers before us who have written fast and efficient libraries. Java's math package provides the `java.math.BigInteger` class which is immensely useful for this case. Python on the other hand has an `int` type that supports arbitrary precision by default! C++ however does not have a way of handling arbitrary precision in its Standard Template Library. There are also many other implementations of arbitrary precision arithmetic online, and you are free to use these as long as they are published before the contest.

Now that we can represent $\mathbf{N}$, we can simply use the inverse sum formula discussed in the previous section to compute the answer. Arithmetic on integers of this size is relatively fast, but the bottleneck in this approach is in the computation of the integer square root. However, if you are using Python or Java, you can use `math.isqrt` and `java.math.BigInteger::sqrt` as these highly optimized library functions will handle the heavy lifting for you.

In general, if we decide to not use a library method, there are many algorithms for computing square roots. Some popular ones include the Babylonian method or the shifting nth root algorithm. For this problem, most algorithms with a complexity of anywhere between $O(log^2(n)log(log(n)))$ and $O(log^{2.58}(n))$ should be able to pass provided your constant factor is sufficiently small.

There's a lot more we were unable to cover in this analysis, but if you find numerical or mathematical algorithms interesting, we highly recommend you explore this area out! The Wikipedia page for the computational complexity of mathematical operations is a good place to start with a lot of interesting approaches to various well-known problems.