CALSUCO '22 Editorial

July 30th, 2022
120 Minutes
4 Problems
Editorial by CALICO

Table of Contents

Problem Name	Page	Points
Problem 1: Ice Cream Bars!	3	5+4+4 = 13
Problem 2: Rock Paper Strategy!	8	0+10+12+14 = 36
Problem 3: Making the title for this programming problem was a lot of work	12	13+8 = 21
Problem 4: CALICO's Corporate Conundrum	16	15+9+6 = 30
Credits	21	
Total Possible Points		100

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.

Problem 2: Rock Paper Strategy!
0+10+12+14=36 Points

Problem ID: `rps`
Rank: `0+1+2+3`

Introduction

Problem Summary

Play rock paper scissors against a bot whose source code has been given to you. In each round, you output your move first before reading in the bot move, but you can assume the bot does the same and you're basically playing simultaneously.

Each round, the bot first uses logic depending on the last two moves played by both players to decide what move to play. If that fails, it falls back to letting a pseudorandom number generator decide instead.

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

Big Ideas

While the main test set can be solved with randomization or a simple pattern, the first bonus required some logical reverse engineering, and the second bonus required some number theory to crack a linear congruential generator.

Linear congruential generators, or LCGs are a very popular type of pseudorandom number generator! They're simple, fast, and the default random algorithm used by many languages. But the quality of their randomness can sometimes be poor, which leads to them being exploited like in this problem.

If there's one thing you should take away from this problem, never use an LCG for cryptographic or security purposes.

Main Test Set

To pass this test set, we need to achieve a win rate of 30%. This can actually be done using a strategy that doesn't even consider the bot's moves!

Roll the Dice *Solutions available in Java, Python*

The game theory optimal strategy for rock paper scissors is to always play a random move with equally likely probability. This can be done by picking a random number between 0, 1, and 2 using `random.randrange` in Python, `java.util.Random::nextInt` in Java, or `std::rand()` and modding by 3 in C++. Then we decide to play `R`, `P`, or `S` depending if the value is `0`, `1`, or `2`. It's easy to see that this yields an expected $\frac{1}{3}$ win rate, and when run 10^5 times, it is almost certain to exceed 30%. Although there is no better strategy in general, it's possible to do much better if we instead exploit the bot's strategy specifically.

Finding a Pattern *Solutions available in C++, Java*

Notice that the bot's strategy will always first check if our last two moves are the same. If so, then it will play the move that beats our last move. We can exploit this logic by designing a pattern that tricks the bot so that playing the move that beats our last move is always a losing move! For example, we can play `RRSSPPRRSSPPRR...` because after seeing us play `R` twice, the bot will play `P`, only to be defeated by our `S` next move. Then, after seeing us play `S` twice, the bot will play `R`, only to be defeated by our `P` next move, and so on. This guarantees we at least win every other game, yielding a win rate of ≥50% which exceeds 30%.

There are also other simple patterns that can achieve a 30% win rate or better without even considering the bot's actual moves! We challenge you to try finding some of these on your own.

Bonus Test Set 1

To pass this test set, we need to achieve a win rate of 60%. This can be done by exploiting all aspects of the logic in the bot's main function.

Counter Logic Gaming *Solutions available in C++, Python*

Since the logic depends on our previous two moves and the bot's previous two moves, it'll be useful to keep track of them. We can do this by storing all our moves and all the bot's moves into an array, and keeping track of the current round number.

Now, for every round, we can simulate the bot's logic using the previous moves we've stored. If our last two moves are the same, we can play the move that beats the move that beats our last move. Else, if the bot's last two moves are the same, we can play the bot's last move (since the bot's last move beats the move that beats the move that beats the bot's last move). Else, we can default to playing a random move as well.

The calculation to find the expected win rate is a bit involved so we won't be covering it here. It can be solved by setting up a system of equations or using a Markov chain. However, by running it against the practice test set, we see that this achieves a win rate of around 62%, which exceeds 60%.

Forcing the Hand *Solutions available in C++, Java, Python*

Although it wasn't necessary to pass the test set, we can achieve a slightly better win rate by instead defaulting to playing the last move we've played instead of a random move. This is no better for us on that turn since the opponent's move is random, but it guarantees that the next move played by the opponent is exploitable since our last two moves are the same. This brings our win rate up to about 72%.

Bonus Test Set 2

To pass this test set, we not only need to exploit all of the bot's logic, but also crack the linear congruential generator it uses when it decides to play a "random" move.

A Trit of Number Theory *Solutions available in C++, Python*

The bot's `random_move()` function works by taking X, effectively converting it into base 3, then using the digits to play the next 19 random moves in big endian. When all 19 moves are used up, it calculates the next X value using `X = (A * X + C) % M`.

If only we can figure out the values of X, A, and C, then we would be able to predict the exact "random" move performed by the bot by simulating its source code ourselves. However, the bot retrieves X, A, and C from random.org during runtime, so we can't know beforehand.

Every time the bot plays 19 random moves, we can reverse the base 3 digits to recover a prior X value. We can tell if a move is random by tracking its logic like in the previous section.

Notice that calculating X is performed under a Galois field: the set of integers mod M, as $M = 10^9 + 7$ is prime. After finding just 3 X values using 57 random moves, we can use modular arithmetic to solve all future values of X, A, and C. Call these values X_0, X_1, and X_2. We know:

$$X_1 \equiv AX_0 + C \bmod M$$

$$X_2 \equiv AX_1 + C \bmod M$$

Subtracting these equations, we get:

$$X_2 - X_1 \equiv AX_1 + C - AX_0 - C \bmod M$$

After canceling the Cs out, we collect the As with the distributive property. We can now solve for A by taking the modular inverse of (X_1 - X_0) under M using the extended Euclidean algorithm or using binary exponentiation with Fermat's little theorem and multiplying by it on both sides:

$$X_2 - X_1 \equiv A(X_1 - X_0) \bmod M$$

$$A \equiv (X_2 - X_1)(X_1 - X_0)^{-1} \bmod M$$

Knowing A, we can substitute it into any of our original equations to solve for C. We now have all we need to predict and counter every future move made by the bot with perfect accuracy!

Although we cannot always win on the first 57 random moves, every move after that is guaranteed. Thus, our win rate over 10^5 games is >99.943%, which exceeds the required 90%.

Problem 3:
Making the title for this programming problem was a lot of work
13+8=21 Points

Problem ID: `haiku`
Rank: `2+3`

Introduction

Problem Summary

Given $\mathbf{N}$ words and their syllable count, construct a haiku: a three line poem with words containing 5 syllables in total on line one, 7 on line two, and 5 on line three.

All words are unique and can only be used once. If it's impossible to construct a haiku with the words given, output `IMPOSSIBLE` on all three lines instead.

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

Big Ideas

The main test set can be solved by trying every possible haiku in factorial or exponential time. However, the bonus required the use of dynamic programming to optimize the exponential time solution down to linear, or we can take a completely different approach by examining integer partitions.

The problem introduction features Anya and Damian from the anime Spy x Family which recently finished airing! You can also legally stream it on Crunchyroll.

Main Test Set

We only have 9 words for this test set. This makes multiple brute force approaches viable.

Poetry Recital *Solution available in Python*

When reciting a poem to a friend, you basically just say all of the words in order out loud! We can take inspiration from this simple fact and have our program do something similar.

We can enumerate through every permutation of all words, and for each permutation, go through the words in order and see if the word breaks align with the syllable breaks at 5, 12 (5 + 7), and 17 (5 + 7 + 5) syllables. Python's `itertools.permutations` and C++'s `std::next_permutation` provide easy and efficient ways to do this.

If we find a permutation that works, we can use the word breaks to construct our haiku. If not, then there must be no way to construct a haiku, so we conclude it's impossible.

As we need to consider $\mathbf{N}!$ permutations and need $O(N)$ time to check each permutation, our overall runtime is $O(N! \cdot N)$.

Chomsky Who? *Solutions available in C++, Python*

Since we don't care about grammar, the order in which words occur on each line doesn't matter at all! We can use this observation to speed up our program. This wasn't necessary to pass the main test set, but it helps us see an even bigger optimization in the next section.

Whereas before each word has $\mathbf{N}$ different positions to be in within each permutation, observe that only 4 of these are relevant: in the first line, in the second line, in the third line, or in no line.

Implementing this solution can be done easily by looping up to `1 << (2 * N)` and using a bitmask. If you're using Python, `itertools.product` can also help with implementation. If you really want to, you can also write an octuple nested for loop.

For each word, we need to consider which of the 4 options to assign it. Since we need to do this for all words, we need to check $4^{\mathbf{N}}$ assignments in total. For each assignment, we can do constant time arithmetic to verify if the number of syllables on each line is correct. Thus, our overall runtime is $O(4^N)$.

Bonus Test Set

With 1000 words, exponential time is out of the question. We need something much faster.

The Reduction is Left as an Exercise to the Reader *Solution available in Python*

Some of you may have noticed that this problem is quite similar to the subset sum problem (SSP), which is a well-known NP-complete decision problem in computer science. It's not too difficult to see that since our syllable totals per line are small, we might be able to take some inspiration from the dynamic programming solution to SSP.

The difference, however, is that we need to consider additional states in order to represent the different number of syllables in each line. One way to do this is to represent states as quadruplets in the form of (i, a, b, c) where i is the index of the current word being considered, a is the number of syllables already in the first line, b is the syllables in the second, and c is the syllables in the third. For each state, we can either add word i to any of the lines or skip it.

If some state $(i, 5, 7, 5)$ is achieved, then we know it must be possible to construct a haiku. We can follow the transitions to recover which words were assigned to each line to construct our final answer. If $(i, 5, 7, 5)$ is never achieved, we know it must be impossible.

Since we have $N \bullet 6 \bullet 8 \bullet 6$ states to consider and computing the transitions from each state can be done in $O(1)$, our overall runtime is $O(N)$. By default, our space complexity is also $O(N)$ but this can be reduced to $O(1)$ by performing a bottom-up optimization similar to SSP.

Cutting the Crap

However, with $\mathbf{N}$ going up to 1000, considering $1000 \bullet 6 \bullet 8 \bullet 6 = 288000$ states may be too slow despite being asymptotically optimal depending on your language and other constant factors. We can get around this by realizing that most words in the list are actually useless! We'll never have a use for our 8th 2-syllable word because if we used all 7 prior, we would have run out of syllables to use the 8th. We'll never have a use for our 18th 1-syllable word, 5th 3-syllable word, 4th 4-syllable word, 4th 5-syllable word, 2nd 6-syllable word, 2nd 7-syllable word, or any words with more syllables for the same reason. Thus, we really only need to consider at most $17 + 7 + 4 + 3 + 3 + 1 + 1 = 36$ words or at most $36 \bullet 6 \bullet 8 \bullet 6 = 10368$ states. This drastically cuts down our run time in practice.

Pristine Partitions *Solutions available in Java, Python*

Another approach we could also take is to observe that there just aren't a lot of ways to make a 5 or 7 syllable line using words! Since we've already established the word order in each line is irrelevant, it's only the number of words with each number of syllables that are relevant to each line. The syllable structures are given by the integer partitions of 5 and 7.

There are only 7 ways to make a 5-syllable line. These syllable structures are:

```
[5]                 [4, 1]              [3, 2]
[3, 1, 1]           [2, 2, 1]           [2, 1, 1, 1]
[1, 1, 1, 1, 1]
```

Likewise, there are only 15 ways to make a 7-syllable line. These syllable structures are:

```
[7]                 [6, 1]              [5, 2]
[5, 1, 1]           [4, 3]              [4, 2, 1]
[4, 1, 1, 1]        [3, 3, 1]           [3, 2, 2]
[3, 2, 1, 1]        [3, 1, 1, 1, 1]     [2, 2, 2, 1]
[2, 2, 1, 1, 1]     [2, 1, 1, 1, 1, 1]  [1, 1, 1, 1, 1, 1, 1]
```

These syllable structures could be programmatically generated, or even found by hand.

We can choose any 5-structure, 7-structure, and another 5-structure to create a full haiku structure. There are only $7 \bullet 15 \bullet 7 = 735$ haiku structures we need to check.

With this in mind, we can check if a haiku can be constructed by checking if it's possible to use our words to fulfill any haiku structure! This can be done very cleanly by using a table to count how many words of each syllable we have, then checking each structure to see if our table has at least as many words for each syllable in the structure. If the structure can be filled, we construct a haiku using that structure and any words with the corresponding syllable counts.

It should be easy to see that this simple algorithm runs in $O(N)$ time and uses $O(1)$ space.

Cutting the Crap… Again *Solution available in Python*

Although it wasn't necessary for this test set, it's possible to cut down the number of structures by only considering haiku structures with unique total syllable counts. For example, consider `[[4, 1], [5, 2], [3, 1, 1]]` and `[[5], [4, 3], [2, 1, 1, 1]]`. By ignoring these symmetries, we are left with a mere 155 unique structures that need to be checked.

Problem 4: CALICO's Corporate Conundrum
15+9+6=30 Points

Problem ID: `managers`
Rank: `2+3+3`

Introduction

Problem Summary

Given the managers of a company's $\mathbf{N}$ employees, find the maximum number of unique disputes a single employee is responsible for resolving. A dispute between two employees is resolved by the lowest-level manager that has authority over both of them, directly or indirectly.

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

Big Ideas

It's been a while since we had a graph problem, so we hope that this one was a welcome addition to the contest!

Because every employee has exactly one manager, and the company must be connected, we know that the company must be a tree with the CEO at the root and manager-subordinate relations forming the parent-child edges. For the main test set and first bonus, the fact that each employee manages at most two others tells us we're working with a binary tree specifically.

Although a brute force approach was sufficient for the main test set, the step up to the first bonus required a clever change in approach for efficiency, and the step to the second bonus required a small formula optimization.

Main Test Set

Brute Force *Solution available in Python*

We can pass the main test set by identifying all possible disputes between pairs of employees. Then for each dispute, find the lowest-level manager that has authority over both of them and count the number of disputes that manager is responsible for resolving.

One way to find the lowest-level manager with authority over two employees is to:

1. Follow the chain of management upwards from one employee until you reach the CEO—in other words, find the manager of the current employee, then find their manager, etc. until you reach the CEO.
2. Track each employee you see in a set (including the original employee).
3. Then, do the same for the other employee, until you find an employee shared by the first employee's chain of management.
4. The first employee you find using this method will be the one to resolve their dispute.

After finding the resolving manager for all pairs of employees, we output the maximum number of disputes a single employee is responsible for resolving.

Since there are $N(N-1)$ pairs of employees in the company, the number of unique disputes within the company grows at a rate of $O(N^2 - N) = O(N^2)$. Each pair then requires the calculation of who will resolve the dispute. The complexity of the above implementation depends on the height of the management tree. For a typical, balanced binary tree, this grows at a rate of $O(log(N))$; however, in the worst case, the management tree could be a straight line—where every employee manages at most one other employee—so the height could grow at a rate of $O(N)$. Therefore, this brute force approach would have an overall runtime of $O(N^3)$.

Lowest Common Ancestor

Those of you more familiar with the subject may have recognized finding the lowest-level manager as the lowest common ancestor (LCA) problem. While algorithms to quickly compute the LCA exist, the bottleneck for this particular approach is the need to compute $O(N^2)$ different LCAs. That being said, it's possible to design a $O(N^2 log(N))$ or even $O(N^2)$ optimized brute force algorithm using these methods, although $O(N^3)$ was sufficient for this test set.

Bonus Test Set 1

Now that we have 10^5 employees, we need a faster way without looking at all employee pairs.

Paradigm Shift *Solutions available in Java, Python*

In the previous test set, we approached the problem by finding who's in charge of resolving any given dispute. Flipping our approach to this problem—finding which disputes can be resolved by any given employee—gives the potential to yield considerable efficiency advantages.

Let's define the branch of an employee as everyone they have authority over, including themselves. Every employee has a branch of the company in which they lead.

We need to recognize that a dispute will only be resolved by an employee if either:
1. the employee themself is one of the parties involved and the other party is a subordinate, or
2. both disputing parties are on different branches of the employee's direct subordinates.

The number of disputes that fall under the first condition is equal to the number of subordinates (direct or indirect) an employee has. In other words, the branch size of that employee minus 1. The number of disputes that fall under the second condition is equal to the size of the left branch multiplied by the size of the right branch—or zero, if the employee has less than two subordinate branches. Therefore, efficiently calculating employee branch sizes is key.

We can find these sizes recursively, where the size of an employee's branch is the size of its left and right subordinate branches added together (if present), plus one. Employees with no subordinates would have a branch size of one. While recursing, we can calculate both the number of subordinates an employee has, as well as the number of employees in each of their subordinate branches. With this information, we can calculate the number of disputes each employee needs to resolve, keeping track of the maximum along the way.

This recursive approach is known as depth-first search (DFS), and it lets us calculate the number of subordinates underneath each employee—alongside the number of disputes they're responsible for—in linear time $O(N)$.

Note that multiplying large branch sizes can result in values of up to $10^5 \bullet 10^5 = 10^{10} > 2^{32}$, so we should use a data type capable of handling larger numbers (such as a 64-bit integer).

Bonus Test Set 2

With employees able to have more than two subordinates, we need to generalize how we calculate the number of disputes under the second condition, where both disputing parties are on different subordinate branches.

With at most two branches, this used to mean multiplying the number of employees in each branch together. With more than two branches, we need to recognize this means (efficiently) finding the sum of all pairwise products between subtree sizes.

Optimized Summation *Solution available in C++*

Suppose we have subordinate branch sizes $x_1, x_2, \ldots, x_m$ for a given employee. Since we want to find the sum of pairwise products between different branches, we're essentially looking for:

$$\sum_{i \neq j}^{m} x_i x_j = \sum_{i=1}^{m} \sum_{j=i+1}^{m} x_i x_j$$

Computing this directly with a double loop would cause this calculation to have a runtime of $O(m^2)$, due to there being $m(m-1)$ pairs. This would cause our solution's runtime to increase to $O(N^2)$, which would be too large of an efficiency loss.

It's easier to see how to optimize this calculation if we unroll the sums. Doing so, we get the sum on the left. If we apply the distributive property to each row, we get the sum on the right:

$$\begin{aligned} x_1x_2 + x_1x_3 + x_1x_4 + x_1x_5 + \ldots + \\ x_2x_3 + x_2x_4 + x_2x_5 + \ldots + \\ x_3x_4 + x_3x_5 + \ldots + \\ x_4x_5 + \ldots + \\ \ldots \end{aligned} \qquad \begin{aligned} x_1(x_2 + x_3 + x_4 + x_5 + \ldots) + \\ x_2(x_3 + x_4 + x_5 + \ldots) + \\ x_3(x_4 + x_5 + \ldots) + \\ x_4(x_5 + \ldots) + \\ \ldots \end{aligned}$$

It's easy to see now that we can efficiently compute this sum by precomputing the right hand side of each row. Begin by setting RHS to $x_1 + x_2 + x_3 + \ldots + x_m$. Then, for each branch x_i, subtract x_i from RHS and add $x_i \cdot RHS$ to the total.

The calculation for each employee's disputes now takes $O(m)$ operations, but since the sum of all m never exceeds $\mathbf{N}$, our overall runtime remains $O(N)$.

Another way to approach this calculation is as follows: if we let S represent the sum of all branch sizes (in other words, $S = x_1 + x_2 + ... + x_m$), then we can recognize that each member of the ith branch will have $S - x_i$ possible disputes in other branches. We subtract x_i from S because we only want to include employees from other subtrees in the dispute; otherwise, this would cause a different employee to become responsible for resolving it.

One thing to note is that we want to avoid double counting pairs—a dispute between employees A and B should not be seen as distinct from a dispute between B and A. As a result, we divide by two to find the number of distinct disputes, giving the formula:

$$\frac{1}{2}\sum_{i=1}^{m} x_i(S - x_i).$$

This approach also has an overall runtime of $O(N)$, similar to the previous approach.

Negative Space

Yet another approach to computing the same sum is to recognize that it is equivalent to subtracting the total number of pairs within each subordinate branch from the total number of pairs in the employee's subordinate branches.

To find the total number of pairs in the employee's subordinate branches, we can start by finding S, the sum of each branch size. The total number of pairs then is given by $\frac{1}{2}S(S - 1)$. Similarly, the total number of pairs within each subordinate branch x_i, is given by $\frac{1}{2}x_i(x_i - 1)$. Like the last section, we divide both quantities by 2 to prevent double counting.

Thus, our final formula is:

$$\frac{1}{2}S(S - 1) - \sum_{i=1}^{m}\frac{1}{2}x_i(x_i - 1)$$

This approach also has an overall runtime of $O(N)$, similar to the previous approach.

Similarly to the last test set, we should use a data type capable of handling larger numbers (such as a 64-bit integer).

Credits

Contest organized by Chris Liu and Ryan Zhu.

Ice Cream Bars!

Written, prepared, and solutions by Ryan Zhu.
Editorial by Ryan Zhu and Aleksandar A. Yanev.

Rock Paper Strategy!

Written, prepared, solutions, and editorial by Ryan Zhu.

**Making the title
for this programming problem
was a lot of work**

Written and prepared by Ryan Zhu.
Solutions by Ryan Zhu and Chris Liu.
Editorial by Ryan Zhu.

CALICO's Corporate Conundrum

Written and prepared by Chris Liu.
Solutions by Chris Liu and Ryan Zhu.
Editorial by Aleksandar A. Yanev, Chris Liu, and Ryan Zhu.

Special Thanks to:

Aleksandar A. Yanev, Emma Fu, and Anthony Chang for proofreading and providing feedback for the editorial.
Xavier Plourde for testing the problems on the judge platform.
Manuel Calva Hernandez for contributing formatted solutions in C++.

… and all of you for taking the time to participate in our contests and reading our solutions!