Florida High School Programming Series
2017 Playoff Round

Problem Writer's Analysis by Arup Guha

Problem A: At Least Average

The most nieve way to solve this problem is to loop over all intervals[i, j], and for each interval, run a loop to get the sum of the elements, and divide by the length of the interval. Unfortunately, this nieve approach has a run time of $O(n^3)$, where n is the size of the list and in this problem, the list could be as large as 10^5. 10^{15} operations is much, too much for a programming contest problem.

A natural optimization is taking the sum of the interval [i, j] and adding array[j+1] to it to obtain the sum of the interval [i, j+1]. Using this optimization, we reduce our runtime to $O(n^2)$. In code, it might look something like this:

```
public static long solve(int[] list, int ave) {
    long res = 0;
    for (int i=0; i<list.length; i++) {
        int sum = 0;
        for (int j=i; j<list.length; j++) {
            sum += list[j];
            if (sum >= (j-i+1)*ave)) res++;
        }
    }
    return res;
}
```

But, again, for programming contests, 10^{10} operations, even if they are fairly simple, are too much. In fact, notice that the final answer might be close to 10^{10}. Thus, any solution that adds 1 at a time is too slow. Thus, we need some mechanism to allow us to add more than 1 to our final tally at a time.

A Fenwick Tree (or Binary Index Tree) is a data structure that allows the two following operations on an array with indexes [1..n], both in O(lg n) time:

1) Add some value, v, to array index i.
2) Query the sum of any interval [i, j].

So, for example, if we had added 1 in a Fenwick tree to the indexes 3, 6, 3, 4, 5, 1, 2, 4, 3, and 5, and we wanted to know how many values greater than or equal to 4 we added to it so far, we could quickly get an answer of 5.

Upon first glance, it's not clear how a Fenwick tree helps us for this problem.

First, let's consider obtaining the sums of every subsequence starting from the beginning - we do this by reading in the data, and then running a for loop so that the data is stored in a cumulative frequency array.

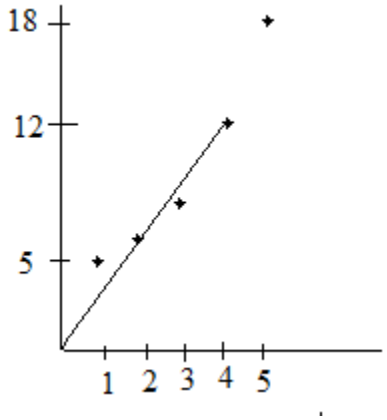
A cumulative frequency array cf of an array arr simply has cf[i] = arr[0] + arr[1] + arr[2] + ... arr[i].) Given a regular array, here is how we turn it into a cumulative frequency array:

```
for (int i=1; i<arr.length; i++)
    arr[i] += arr[i-1];
```

Now, in terms of this problem our array stores the player's total score after each round. Consider the example given in the problem text:

Original values: 5, 1, 2, 4 and 6
Cumulative frequencies: 5, 6, 8, 12 and 18

Notice that the difference between any two of the cumulative frequencies gives the sum of a range. For example, 18 - 6 = 12, gives us the sum of the last three items of the original list, since 18 represents the sum of the first 5 items and 6 represents the sum of the first two items. When we subtract these, we are left with the sum of the last three. Consider the problem of determining how many of the given intervals ending in index 3 in this example have an average of at least 3. Visually we have:



The line drawn in has slope 3. When considering the point (4, 12), we'd like to know how many points drawn to the left of it are at or below the line, since these represent a slope (ie average during a span of levels) of 3 or greater. Unfortunately, although a Fenwick tree can tell us how many of the points to the left of (4, 12) are greater than or equal to 12, or less than or equal to 12, it can't tell us how many points to the left of (4, 12) are below that given line.

BUT...what if we were to add offsets to each of our values??? In particular, rather than arr[i] being our score after level i, what if we *pretended* to play the rest of the game from that point on with an average of ave??? So, for example, after level 0 (first level), we always scored 3 for the remaining levels. If we did, we'd have a score of 5 + 4(3) = 17. Do the same for each of the input values to get the new following list:

Cumulative frequencies: 5, 6, 8, 12 and 18
Cumulative frequencies with offsets: 17, 15, 14, 15, and 18.

Now, when we consider the fourth value in the list 15, what we care about is how many values before it are 15 or less. In this case, there are 2 such values: 15 in index 1 and 14 in index 2. A Fenwick Tree CAN answer this query quickly!!! In short, if a number 15 or less appears to the left of the 15 we are considering, then that means the difference between the two corresponding values in the cumulative frequency array is at least ave*(j - i + 1). This in turn, indicates a set of consecutive levels with an average of at least ave. For example, in index 3 we have 15 = 12 + 3 and in index 1 we have 15 = 6 + 3*3, when we subtract these two, we get 12 + 3 - (6 + 3*3) = (12 - 6) + (3 - 3*3) = (12 - 6) - (3 - 1)*3. Basically, the first part of this is the actual difference between the cumulative frequencies - the sum of levels 3 and 4 and the second portion of this is subtracting out what would be the sum of this interval if a player scored 3 each time.

Thus, once we write a Fenwick Tree, the heart of our solution looks like this:

```
long res = 0;

for (int i=0; i<=n; i++) {
    res += myBit.sum(vals[i]+1);
    myBit.add(vals[i]+1,1L);
}
```

My Fenwick Tree is 1 based, so I add 1 to each item in my cumulative frequency array, which is stored in vals. Also, in my cumulative frequency array, I prepend a value of 0 and make the array size n+1 so that all intervals that start at the beginning can be represented as the subtraction of two elements of the cumulative frequency array.

To learn more about Fenwick Trees, check out the following tutorial on TopCoder:

https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

Problem B: Bonus Points

This question is intended to be a straight-forward simulation, where students must implement a given set of rules exactly. The key is simply to maintain a variable for the total score and a second variable for the current bonus level. After adding a level score, simply check to see if the total score exceeds what's needed to get the bonus. If it is, continue doubling the bonus level until you get to the proper bonus level (since a player may skip several bonus levels with a really good round). Then, simply add this bonus to their score. Of course, if after adding a level score, the total score doesn't exceed what's needed to get the bonus, then simply move onto the next level.

Problem C: Counting Factors

There are at least two different strategies to solve this problem (and probably more). For this write up, I'll discuss a strategy that easily passes the constraint $n \leq 10^4$, but doesn't necessarily pass the $n \leq 10^{10}$ bound.

The first fact that is necessary to solve this problem is knowing the number of factors an integer has, given its prime factorization. An integer of the form $n = p^a q^b r^c$, where p, q and r are distinct primes has precisely (a+1)(b+1)(c+1) factors. (This formula generalizes to any number of distinct prime factors.) One way to see this formula is to note that all factors of n must have the form $p^x q^y r^z$, where $0 \leq x \leq a$, $0 \leq y \leq b$, and $0 \leq z \leq c$. Notice that there are precisely (a+1) possible choices for x, (b+1) possible choices for y and (c+1) possible choices for z. The choice of each exponent is independent of the others, so to get the total number of possibilities, we multiply.

Now, let's consider trying to find the number of values with Viraj prime factorizations that have 48 factors. What we're really looking for is some exponents so that each exponent plus 1 multiplies to 48. Consider one such product: 2 x 2 x 2 x 6. We must have three different primes, each raised to the first power and one prime raised to the fifth power. Since there are 25 primes less than 100, we have $\binom{25}{3}$ ways in which we can choose the primes raised to the first power for our Viraj prime factorization. Then, we have 22 distinct primes that remain and can choose any one of them (in $\binom{22}{1}$ ways) for our prime factorization. Thus, there are $\binom{25}{3}\binom{22}{1}$ Viraj prime factorizations of the form $pqrs^5$, where p, q, r and s are distinct primes.

In our solution however, we won't individually go through each possible factorization in this manner. Instead, we will solve the problem recursively, only keeping track of what we need to as we build the solution.

Again, consider the case with n = 48. Our first term in our product can be a 2, 3, 4, 6, 8, 12, 16, 24 or 48. (These correspond to exponents 1, 2, 3, 5, 7, 11, 15, 23 and 47, respectively.) Each of these will spawn recursive calls and we'll add the results of all the recursive calls as we've split up our prime factorizations into these categories.

If the first term of our product is 2, then naturally, the product that remains to be formed is 24. But, if we allow any recursive chain of products, we'll double count products in different orders. Thus, we must force our products to go in increasing order, to prevent double counting. Also, note that 2 can appear more than 1 time in our product - it can appear up to 4 times. Thus, within our case for 2, we have the following possibilities:

current product = p, remaining value of n = 24
current product = pq, remaining value of n = 12
current product = pqr, remaining value of n = 6
current product = pqrs, remaining value of n = 3

Recall that each of these products can be done in $\binom{25}{k}$, where k is the number of primes appearing in the current product at this first breakdown. Let our recursive function be defined as follows, mathematically:

f(n,min,pLeft) = the number of ways to achieve a product of n where each term is strictly greater than min, and the number of primes left (terms in the product) is equal to pLeft.

Just this portion of the answer can be broken down as:

$$\binom{25}{1} f(24,2,24) + \binom{25}{2} f(12,2,23) + \binom{25}{3} f(6,2,22) + \binom{25}{4} f(3,2,21)$$

Essentially, if we take one prime, we can do so in 25 ways and we must multiply what's left to 24, each term being strictly greater than 2, and there are 24 more primes to use up. If we take 2 primes each to the first power, we can do this in 25 choose 2 ways and we multiply what's left to 12, each term being strictly greater than 2, and now there are 23 more primes left to use, and so forth. Next, this breakdown has to be done for 3, then 4 etc. and ALL of these recursive call results added up and modded.

Here is the key portion of my code that adds all of this up:

```
for (int i=0; i<numD; i++) {
    int div = divisors.get(i);
    if (div <= lastExp) continue;
    if (n%div != 0) continue;
    int tempn = n/div;
    for (int j=1;; j++) {
        res = (res + combo[primesLeft][j]*go(tempn, div, primesLeft-j))%MOD;
        if (tempn%div != 0) break;
        tempn /= div;
    }
}
```

Problem D: Defeating Dragons

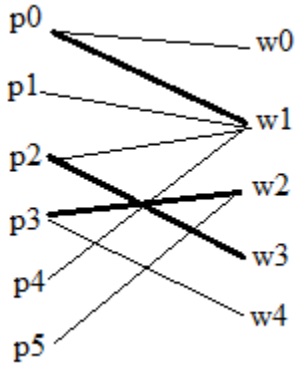
This problem is probably not doable for students who haven't been introduced to network flow or bipartite matching. It's an alluring problem to try as many greedy solutions seem to yield optimal solutions for many small cases, but all of these will fail more complicated cases.

The idea for the solution is as follows:

Create two lists, one for the poisons and one for the weapons:

Poisons	Weapons
p0	w0
p1	w1
p2	w2
p3	w3
p4	w4
p5	

Now, for each dragon, draw a line connecting the poison that kills it to the weapon that kills it. Here is a sample drawing, with a few lines bolded in (the explanation for these will be given later...)



Notice that none of the bold lines share any end points. That means that for the three dragons those three lines represents, we can't get a "two for one". Namely, each of those three dragons must be killed with a different poison or weapon. Thus, those lines prove that we need at least 3 poisons or weapons. Thus, in some sense, our goal is to draw a maximal number of these lines that don't share endpoints. In this particular drawing, there does exist an arrangement with four matching poisons and weapons with no overlap: (p0, w0), (p1, w1), (p2, w3), and (p3, w4). (And there are others.) But, no arrangement exists where 5 separate lines can be drawn across. Thus, for the query modeled by the drawing above, 4 poisons or weapons would suffice.

If we add two special vertices to the picture above: a source vertex to the left connecting to all poisons and a sink vertex that all of the weapons connect to, then the maximum flow through that flow network (where each edge has weight 1), is equal to the maximum number of poisons and weapons one would have to carry to slay all dragons. (Network Flow is too involved to discuss fully here - the most simple version requires multiple iterations of depth first search. A more efficient version requires a combination of breadth first and depth first search.)

Problem E: Electric Car Race

Since there are no more than 10 stops, we can simply try all permutations (each equally likely according to the problem statement), and see how many of them allow Pamela to finish the race. Once we determine this number, our result is this number divided by n!, where n is the number of stops. To display the final answer, we must take this fraction and divide both the numerator and denominator by the greatest common divisor of both numbers, to get a fraction in lowest terms.

When we try each permutation, it's best to keep the permutation code separate from the code storing information about the stops. Namely, never swap around values that are stored about each of the stops. Instead, keep a separate array storing the integers 0, 1, 2, ..., n-1, and have these values swap around. Then, use this array to index into the array storing the data for the problem.

For a fixed permutation, to simulate if it will work or not, simply have a variable that keeps track of current charge. Simulate traveling in the given permutation, subtracting out the distance traveled between consecutive stops from the current charge. If this yields a negative number, immediately return false, indicating that this permutation won't result in the race being finished. If we arrive safely at a point, check to see if it's a charging station. If so, update the current charge to full capacity.

While one could create a fraction class, it's easy enough to just count the number of permutations and when displaying the result, just divide both the numerator (number of successful ways) and denominator (n!) by the greatest common divisor of the two.

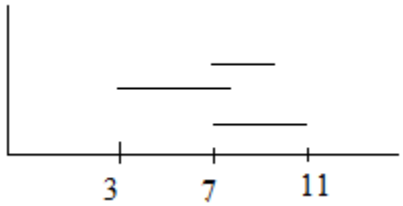
Problem F: Frogger for Four Year Olds

There are a couple different ways to view this game, but the key idea in simplifying the solution is to realize that because all of the prizes move at the same rate, we can calculate where a prize will be for Anya when she gets to that y value. For example, if Anya is moving at 3 units per second and the prizes were moving at 6 units per second, there is one prize starting at (100, 20) and another prize starting at (50, 14), then if Anya were to push her button right away, the first prize would get to (100 + 20*6/3, 20) when Anya gets to y = 20 and the second prize would get to (50 + 14*6/3, 14) when Anya gets to y = 14. In some sense, we can translate each prize which starts at $[(x_1, y), (x_2, y)]$ to $[(x_1 + y*w/c, y), (x_2 + y*w/c, y)]$. After the translation, vertical lines represent Anya moving instantaneously vertically up the screen. Thus, after our translation, we are looking for the vertical line that intersects the maximum number of translated prizes, each each vertical line just represents what Anya will cross for pressing the button at various times. So, the problem reduces to finding the maximal interval cover: given a set of intervals, determine the maximum number of those intervals which cover a particular number. We can solve this by creating events and sorting the events by x value. Each event is either the beginning of an interval or the end of an interval. We assign a value of +1 to the beginning of intervals and a value of -1 for the end of intervals. Process +1s before -1s though this should have no bearing on the final answer because it's guaranteed that the maximal answer exists for .001 seconds. Sort these intervals, by y and then add +1 or -1 for each one as we go along, keeping track of the maximum sum achieved, since that sum at any point represents the current number of intervals open.

Let's consider one example in detail, with the three following prizes:

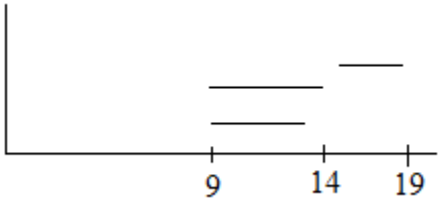
(7, 1) to (11, 1)
(3, 3) to (8, 3)
(7, 4) to (10, 4)

Our initial picture is as follows:



Let Anya's velocity be 3 units/second and the prizes move at 6 units/second. Then, relative to where Anya WILL BE, the translated prizes are:

(9, 1) to (13, 1)
(9, 3) to (14, 3)
(15, 4) to (19, 4)



Now, let's say that the first two prizes are values 10 and 12, respectively and the last prize is value 21. Our events, in sorted order are processed as follows:

1) x = 9, start prize 1, add 10
2) x = 9, start prize 2, add 12, tally = 22
3) x = 13, end prize 1, subtract 10, tally = 12
4) x = 14, end prize 2, subtract 12, tally = 0
5) x = 15, start prize 3, add 21, tally = 21
6) x = 19, end prize 3, subtract 21, tally = 0

Thus, once we translate our values, there's no real geometry going on. We just create a class of these events, sort them, and process them in this manner, sweeping through the events (left to right or right to left), keeping track of our prize tally at any given vertical line.

Problem G: Ground Game

This question is the banger in the set. Keep a variable that tracks how many levels underground the player is and keep a second variable that stores the maximum levels underground the player has gone. When processing the input, ignore characters moving left(<) and right(>), add one to the count when moving down (v), and subtract one to the count when moving up (^). After each change to the counter, see if the new value is greater than the previously seen maximum. If so, update the maximum.

Problem H: High Score

This problem is the custom sorting problem in the set. (I've tried to include one in most years since I think learning how to sort objects in a custom manner is a very valuable skill.) For this problem the intended solution is to create a score object of some sort that stores all of the level scores for a player.

From there, a compare function/method needs to be written that compares two score objects. First, if the number of levels played is different between two score objects, the one with more levels should come before the other player. Next, if two players have the same number of levels played, look at both players' total scores. If these are different, then the player with a higher total score comes before the other player. If these are the same, then we must go through both players' level by level scores, stopping at the first discrepancy. (A while loop probably makes sense here.) Then, whichever player has a higher score in the first level with the discrepancy comes before the other player. Finally, if all the level scores are identical, we can just compare the two input strings (via a typical string compare function/method), with the one coming first alphabetically corresponding to the player that should come before the other player. When outputting the sorted list, simply just print each players' initials.

Problem I: Igloo Construction

This is a classic problem where if we know the outer radius and the desired thickness, we can calculate the volume of ice necessary (by subtracting the larger half sphere volume from the smaller half sphere volume). Since the thickness is fixed, as we increase the outer radius, the volume of ice necessary increases. So, what we can do, is make a guess for the outer radius, see if we have enough ice to build that. If we do great, guess something bigger next time. If we don't guess something smaller. Formally, this technique is known as binary search. If we have a function that is easy to calculate forward that is an increasing (or decreasing) function through its whole domain, but difficult to calculate backwards (given the volume of ice, it's difficult to find the required thickness, or at least it seems at first), then we can run a binary search on the input to the function. We must use the problem bounds to calculate a low and high bound to the possible outer radius. Then, we always guess halfway in between our lowest and highest possible answer. Each guess in the range of [low, high] will result in a new range either [low, mid] or [mid, high], cutting the search range in half. Repeat this 100 times and you're guaranteed to have an answer that is very close to the actual answer.

It turns out that the function involved for this particular question is relatively easy to invert. So, a student who likes mathematics might try this rout. Let r be the outer radius and t the required thickness and V the volume of ice available. We get the following equation:

$$\frac{2}{3}\pi(r^3 - (r-t)^3) = V$$
$$(3tr^2 - 3t^2r + t^3) = \frac{3\pi}{2}V$$
$$3tr^2 - 3t^2r + \left(t^3 - \frac{3\pi}{2}V\right) = 0$$

This last equation is a quadratic in r. We can just solve this for the larger value of r and that's our solution.

Problem J: Just in Case (the set is finished)

In this write up I'll discuss the other solution to counting that does easily work on the $n \leq 10^{10}$ bound. One difficulty with the other solution was that it had to keep track of three things:

(1) value of n
(2) the number of primes left to use
(3) the previous factor used (so future ones were larger)

It seems as if (1) is critical to the problem and can't be avoided. But (3) seems somewhat annoying as does (2). It would be nice if we only had to keep track of extra thing on top of n, instead of 2. The key insight is that there aren't so many Viraj primes to begin with, so instead of skipping some of them by allowing different ones to come earlier in our prime factorization, force every single one to appear in the prime factorization, but allow 0 as a valid exponent!

Now, all the sudden, we no longer need to keep track of our last exponent used! The new exponents, first the one for 2, then the one for 3, and so forth, are 100% independent of the others because they are tied to specific primes!!! So, the only thing we need to keep track of is how many exponents we've already chosen. Think of it this way:

We are filling an array of size 25, call it mult. mult[0] stores the power of 2 (minus 1), mult[1] stores the power of 3 (minus 1), mult[2] stores the power of 5 (minus 1), etc. So, think of a basic recursive function filling up this array and when all 25 slots are filled, we just add 1 to our count.

Now, in our recursive call, all we need to keep track of is

(1) value of n
(2) value of k, the number of slots already filled in

Whenver we fill in array slot k, we just loop through all factors of n (which we pre-compute), including 1, and try it in slot k, adding up our answers for each recursive call.

Now, in this particular form, the recursion will make many redundant calls and time out. To avoid this, just create a two dimensional memo-table (an array to store calculated answers). Before any recursive call completes, store our answer to f(n, k) in memo[n][k]. Then, return the answer. Before any recursive call is made, just see if this particular call has been made before! Of course, n can be very large, so we don't actually index the array with n. Instead, we pre-calculate all divisors of n beforehand (stopping at the square root of n and storing in pairs and then sorting), and assign them a number in their ranked list storing these in a map. We used the mapped value as the actual index to the memo table. (Alternatively, we can store our recursive function results in a HashMap, having no need to index funny.)

Here is my recursive function:

```
public static long go(long n, int primesLeft) {

	int mapNum = map.get(n);
	if (memo[mapNum][primesLeft] != -1)
		return memo[mapNum][primesLeft];

	if (n == 1) return 1;
	if (primesLeft == 1) return 1;

	long res = 0;

	// Try each valid divisor from here.
	for (int i=0; i<divOfDivList[mapNum].size(); i++) {
		long div = (Long)divOfDivList[mapNum].get(i);
		if (n%div != 0) continue;
		res = (res + go(n/div, primesLeft-1))%MOD;
	}

	memo[mapNum][primesLeft] = res;
	return res;
}
```