**FHSPS Round 3: Editorial**

**Comments:**

No technical issues this contest! Something I am personally very happy about.

The problem set turned out to be quite difficult. The easier problems in the set had stricter time limits and were more demanding of students to make observations and create efficient code. Three problems had less than three correct submissions, which was not intended. I will work to make the next problem set more moderate in difficulty.

Onto problem specifics.

**Inversions:**

This problem has a straightforward solution which many teams were able to code quickly, but it involved using nested for loops which led to an **N2** runtime. Since the length of the string can be up to 105, finding inversions by looking at all the characters in the string prior to an index is too slow. Experienced programmers may look at this problem and see it as a classical Fenwick Tree problem, but in truth the data structure is not necessary.

The main observation for this problem is that you can keep a frequency array of length 26, keeping track of the count of each character that already appeared in the string. By iterating through this array at each index (which is at a constant size 26 as opposed to i-1), this allows us to achieve a runtime of O(N).

**Limebike:**

This problem involves a graph where one of the edges gets removed. Let’s say this edge connects nodes a and b. In order to find the number of nodes that are no longer connected, we must first check if removing this edge disconnects a and b. If it doesn’t, print 0 (since b is reachable from a, to every node reachable from a can still reach every node reachable to b). If it does, we must find the number of nodes in the connected components of each node a and b. The answer will be the two numbers multiplied together. All of this can be done by running some graph traversal algorithm such as BFS or DFS. Runtime: O(N+M)

**Marbles:**

This problem relies on an observation: since the marbles are arranged along a circle, all simple polygons made will be convex polygons (the order of which edges are drawn does not matter).

Next, there are no polygons with zero, one or two edges, so we must find the number of ways to make a polygon with three edges, four edges, five edges, and so on. Therefore, the answer is the summation of under mod. There are many ways to compute this, the easiest of which is 2n - - - = 2n - - n – 1.

The runtime of this solution is O(N) (by computing 2n naively).

**Challenge:** Can you solve this problem for an upper bound of n = 1018?

**Octothorpe:**

This problem was almost solved by the team “Definitely Not Russian Hackers,” but in the end it went unsolved. It was one of the hardest problems in the set, as it is difficult to implement correctly.

One observation to be made is that *k*-octothorpes grow in size very quickly, so a 26-octothorpe is already way too big for a query of n = 1018, and every query will always result in ‘.’ This observation is what Definitely Not Russian Hackers were missing, which caused their code to overflow – and when they switched to BigInteger, TLE.

Since *k*-octothorpes are defined recursively, it makes sense to attempt to query an index in it recursively. We mentioned before that *k*-octothorpes grow quickly, so if we decrement *k*, then it will also shrink quickly, vastly reducing the search space.

We need to determine the row and column of *n* in the *k*-octothorpe, and then determine which of the 25 “quadrants” (I use this term in my solution, though it isn’t very accurate since there are 25 original pieces rather than 4) it belongs to. If a row and column “belong” to a quadrant it means that in a (*k*-1)-octothorpe, the current cell was created from that quadrant in one step. Then we must update *n* to be equal to the new cell’s value in the next step. If the quadrant is at the same position as a ‘.’ in a 1-octothorpe, we know that the answer is ‘.’, otherwise we must continue the recursion until the search space is small enough to solve.

By tracing backwards, the size of the octothorpe shrinks by a factor of 25 (5 times less rows and 5 times less columns at each step). We can keep doing this until k = 1 and then simply find the corresponding character in a 1-octothorpe.

If this was difficult to follow, it may be helpful to see the solution on my github page.

Runtime: O(logN)

**Pizza:**

This is the easy problem in the set, and it was solved by every team who submitted compiling code. For this problem, you must sum only the positive differences between two integers of every household. Some teams did not account for the possibility of a household having more pizzas than people (a rare situation in real life), which caused wrong answers. Eventually though, most teams were able to solve this problem.

The runtime of this solution is O(N).

**Quantum:**

This was one of the harder problems, as it was only solved by one team. The problem asks to find the probability that a particle passes through a node by time *k*. This probability can be very small, very large, and it could also change in very tiny amounts at each step. In order to avoid double precision issues, this problem requires you to print the exact probability, but instead of a fraction , it is written as *p*\**q*-1 mod *m*. For this reason, modular inverse is required.

This probability can be computed with dynamic programming. Use two states, representing the probability *p* that Qupee is at node *i* at time *j*. At *j* = 0, the probability that Qupee is at that node is *n*-1. For every 1 ≤ *j* ≤ *k*, *p*[i][j] is the sum of where *y* is each adjacent node to *i* and *num*[*y*] is the number of adjacencies for node *y*.

One strategy to solve this problem is to solve it correctly with doubles first, and then change it to use longs under mod.

The runtime of this solution is O(*nmk*).

**Upvote:**

To solve this problem, we must first convert the time string into an integer. Split the string at the colon ( : ), multiply the hour value by 60 and add the minutes. Keep an array of length 1440 (the number of minutes in a day).

There are two cases:

* The start time is less than the end time - just iterate from start to end and add the number of bots to each time in that interval
* The start time is greater than the end time – in this case there are two intervals: one from the start to the end of the day (1439) and one from the start of the day (0) to the end time. Add the number of bots to every time in these intervals.

All that is left to do is find the maximum value in the array, which can be done by iterating through it.

Runtime: O(N)

**Number of correct solutions for each problem:**

**Pizza:** 34

**Upvote:** 10

**Marbles:** 9

**Inversions:** 8

**Limebike:** 2

**Quantum:** 1

**Octothorpe:** 0