Count The Bits

This problem was a dynamic programming problem based on a function that maps the count of bits and the current value modulo k to a sum of bits. The trick here was realizing that to calculate the sum of bits for values divisible by k, we gain speed by calculating the sum of bits for values equivalent to any j modulo k. That is, we expand the problem somewhat to permit a simple recursive formulation.

We define $f[i][j]$ to be the total number of bits in all numbers with i bits, that are equal to j modulo k. Further, we define $c[i][j]$ to be the count of numbers with i bits that are equal to j modulo k. Clearly $f[0][j]$ and $c[0][j]$ are all equal to zero, except that $c(0,0)$ is equal to one. What happens if we add a zero bit to the right? The modulo value is multiplied by two. What happens if we add a one bit to the right? The modulo value is multiplied by two, and a one is added. Equivalently we can add a bit to the left, which involves tracking the value of that bit modulo k.

So we iterate over the count of bits. From $c[i][j]$ we contribute $c[i][j]$ more bits to $c[i+1][2j \bmod k]$ and also to $c[i+1][(2j+1) \bmod k]$. From $f[i][j]$ we contribute $f[i][j]$ to $f[i+1][2j \bmod k]$, and $f[i][j] + c[i][j]$ to $f[i+1][(2j+1) \bmod k]$.

In the end we read our final result from $f[b][0]$.

Contest Setting

This problem is solved using dynamic programming, with the two dimensional state space mapping the number of difficulty classes considered and the number of distinct problems required into the total number of contests possible.

So we start by calculating how many different problems there are for a given difficulty. We can do this with a hash map. For the third input, which was

```
12 5
3 1 4 1 5 9 2 6 5 3 5 8
```

we end up with the following difficulty counts:

```
1:2 2:1 3:2 4:1 5:3 6:1 8:1 9:1
```

After calculating these counts the actual difficulty doesn't matter at all; we can rearrange this into an array v giving just the counts:

```
[2 1 2 1 3 1 1 1]
```

Define $f(i, j)$ to be the number of contests possible with j different problems using problems from just the first i classes. There's only a single distinct contest with no problems, no matter how many problem classes we consider:

$$f(i, 0) = 1$$

Further, there are no contests possible with at least one problem if there are no problem categories:

$$f(0, j) = 0 \qquad (j > 0)$$

Given a contest with j problems from the first i categories, we can expand it to a contest with $j+1$ problems from the first $i+1$ categories by adding any of the $v[i]$ problems in category i, or we can just ignore that category altogether:

$$f(i+1, j+1) = v[i]f(i, j) + f(i, j+1) \qquad (i, j > 0)$$

We can implement this using recursion with memoization or with iterative dynamic programming.

Coprime Integers

The given limits should have made it clear that calculating the greatest common divisor explicitly for all pairs would not nearly have run in time. Instead, we use inclusion-exclusion on the square-free numbers.

Roughly started, we start with the set of all pairs. We subtract all pairs where both numbers are divisible by 2. We subtract all pairs where both numbers are divisible by 3. We don't need to worry about all pairs divisible by 4, because we eliminated them all when we eliminated all numbers divisible by 2. We subtract all pairs where both numbers are divisible by 5.

Now, think about all pairs where both numbers are divisible by 6. We subtracted them once when we handled all pairs divisible by 2, and once again when we handled all pairs divisible by 3, so we have already subtracted these numbers twice. We need to correct for this by adding back the count of all pairs that are divisible by 6.

And this continues. For every square-free integer k, we either add or subtract all pairs divisible by k. To determine whether to add or subtract, we count the number of distinct prime factors in k; if that number is even, we add; if it is odd, we subtract.

How do we determine how many pairs divisible by k there are with the first number between a and b inclusive, and the second number between c and d inclusive? This is just

$$(\lfloor\frac{b}{k}\rfloor - \lfloor\frac{a-1}{k}\rfloor)(\lfloor\frac{d}{k}\rfloor - \lfloor\frac{c-1}{k}\rfloor)$$

To generate the square-free numbers, we can either test each individually, but that will usually be too slow. Instead, we generate all primes less than a million using a sieve, and then use a simple recursion that either includes or excludes each prime, stopping with the product gets larger than the maximum of b and d. As we generate we can also accumulate the count of primes in each number, so we know whether to add or subtract.

Cops And Robbers

This problem was undisguised minimum-cut (equivalent to maximum flow). Each square gets two nodes, one for entering the square and one for exiting the square, with an edge between them of capacity equivalent to the cost of barricading that square. All squares on the edges evacuate to the sink, and the source was the bank square.

The problem was of sufficient size that a basic Ford-Fulkerson flow algorithm probably would not complete in time, but Edmond-Karp or similar algorithms easily do.

Exam

You want to know how many problems you answered differently from your friend (d), and how many you answered the same e. To maximize your score you allocate as much of your friend's score s to the ones you answered the same (as many as possible), and then the rest to the ones you answered differently. The final result is then

$$\min(e, s) + \min(n - e, n - s)$$

Goat on a Rope

There were multiple ways to solve this problem. One was to enumerate all possible eight cases for where the goat stake could be, and calculate the distance there. Another was to treat the house as a large origin, and essentially compress it to a point.

The problem with enumerating cases is it's very easy to get one wrong.

Ultimately a single line worked for this problem.

```
hypot(max(x1-x, x-x2, 0), max(y1-y, y-y2, 0))
```

Greedy Scheduler

Let's simulate the queue, processing all customers in order from front to back.

At $t = 0$, all of the cashiers are unoccupied and ready to take a customer. Therefore, the first customer will go to cashier 1, occupying that register until $t = t_1$.

In general, say we're about to process customer i. By now we've computed that, due to the shopping carts of customers 1 to $i - 1$, each cashier j is occupied until some time $T(i - 1, j)$, at which point they will become free to take a new customer. Therefore, customer i will go to the cashier j with smallest $T(i-1, j)$, breaking ties by smallest j. As a result, this cashier will become occupied for an additional t_i seconds.

Implementing such a recurrence using saved values is called dynamic programming. It yields a solution with time and memory complexity $O(nc)$, which is sufficient for the provided constraints. The memory complexity can be reduced to $O(n)$ by noticing that there's no need to keep track of $T(i, j)$ for old values of i. Thus, the index i can be suppressed

Finally, we can speed up the search for a minumum time using a partially sorted data structure, such as a balanced binary search tree or min heap. The final run-time is $O(c \log n)$. The limits on this problem were sufficiently relaxed so this was not necessary in this case.

House Numbers

The intended solution here was a simple simulation. Since we are given m, we can simply start with $x = m + 1$ and $n = m + 2$, and accumulate a sum of the numbers from m to $x - 1$ and from m to n. For each x starting with $m + 1$ and increasing by 1, we add $x - 1$ to the sum of houses to the left of us, then increase n (adding it to the sum of the houses to our right) until that sum is greater than or equal to twice the sum to the left of us, plus x. If we hit equality, we are done, else we continue to the next x.

The problem statement guarantees there is a solution that we will find quickly.

Inversions

The unknown values should always be replaced by a nonincreasing sequence. This is easily proved by contradiction. So we just need to find that nonincreasing sequence. A simple dynamic programming approach works, where the state space is (position, rightmost unknown value), working left to right.

The judge solution does it slightly differently; it first calculates the result assuming only the lowest value is available for substitution. Then, using the intermediate results it calculates results assuming the lowest two values are available, and so on and so forth. The state space is the same as the previous approach but the order of computation may be different.

Knockout

The state space for this problem, even starting with a full board, was small enough that a simple recursive solution worked with no need for memoization or dynamic programming. To calculate the expected value, you needed to calculate the probability of each throw and then recursively determine the best actions based on the throw, and return the result up to the top.

One trick to make the same code work for both the minimize and maximize cases was to write the code to maximize the final result, and to compute the minimize case, run it again but negate the final score value in the base case if no result was possible.

Liars

This problem sounds like it requires a complicated solution strategy, but in reality all you need to do is try all the possibilities. For any given number of liars, you can easily check how many people are telling the truth, and see if that is a possible solution. So iterate from 1 to n, check each value to see if it works, and print that value if it does. If no values work, then the statements are mutually inconsistent, so print –1.

Mobilization

Every type of unit can be visualized on a 2D plane with potency vs health (per unit dollar for both). Any linear combination of types of units will lie within the convex hull of these points. The maximum product will also lie on the convex hull, either on an extreme point or on a line of the hull.

So the solution is to compute the hull, and then consider each point and each line in the hull. Optimizing the product of xy on a line in the plane is straightforward calculus, or alternatively it can be done using search.

Interestingly, none of the judge solutions properly dealt with repeated data points, and thus some of the judge data was broken. Luckily this was caught and the data fixed before it affected the contest.

Paper Cuts

This problem was a dynamic programming problem. The function being optimized mapped the used characters in the first string and the final character position from the first string chosen to the fewest number of cuts required to get to that state. For n input characters the total state space was somewhat less than $n2^n$, which for 18 is sufficiently small to easily run in time.

The set of characters already used should be represented as a bitmask in a single integer, rather than a more complicated data structure (like a set of characters); this "bitmap DP" is a standard trick in programming contests.

No additional optimizations were necessary.

Pizza Deal

For both the slice and the whole pizza, simply calculate which has a larger pizza-to-cost ratio. Since the input is integral, and the area of a whole pizza is always π times an integer and thus irrational, it's impossible for there to be a tie. You need to know the formula for the area of a circle given its radius.

Poker Hand

We need to find which rank is represented the most. An easy way to do this is to iterate through all the given cards, and for each one, iterate through all given cards again counting how many match in rank. Return the maximum of this value.

Random Index Vectors

Solving this problem required implementing the operations on Random Index Vectors as described, without expanding the vector to its uncompressed size since that would be too slow. The code that iterated over a pair of RIVs, finding and processing both matching and unmatched indices, is a bit tedious; using functional programming techniques to share this code could simplify the code a bit.

Rectangles

Each vertical line has some x coordinate, so we only need to consider the sweep line right before and right after these x values. For one x value, we multiply the length of the sweep line that had an odd number of rectangles by the previous interesting x value. We then process vertical lines at that position by flipping the parity of some range on the sweep line. We can then continue onto

the next interesting x coordinate.

To simulate these operations efficiently, build a segment tree on the sweep line representing the current parity on each region. Since there are only $O(n)$ interesting y values they could cut the sweep line, there are only n regions. Each vertical line flips some consecutive sequence of regions, which can be done with lazy propagation. We also need to know the length of regions which is odd which can also be maintained by the tree.

Repeating Goldbachs

This problem is just a simulation problem; execute the process described to get the result.

You need a prime sieve to generate the primes less than one million quickly. Then, you just need to scan the primes with two iterators, one increasing and one decreasing, at each step.

The difficult part of this problem was realizing that the process is quadratic if not written carefully. You want to initialize the upper iterator to be the smallest prime not greater than half the current value being split, and you want to initialize the lower iterator to be the largest prime not less than half the current value being split. Scanning the list from either the front or the back for this point is too slow. An easy way to do this is to realize that the required point in the prime list is very close to what it was last time, so searching nearby the most recent point works well.

Time Limits

First, find the slowest solution; this is the one that sets the time limit. You want to calculate the smallest value r that is at least s times the input value, when the input value is given in milliseconds. To calculate this, multiply s times the current input value, and then round up to the nearest second. This can be tricky to get precisely right, but it's not too bad; if the maximum input is v, the result is just

$$\left\lfloor \frac{sv + 999}{1000} \right\rfloor$$