Black and White

The challenge in this problem is writing clear concise code that checks the conditions specified. One trick that can be used is to write code that does the row checks, transpose the input, and rerun that identical code.

Carry Cam Failure

Clearly, with 25-digit input, it's impractical to simply try all possible numbers. But since only the least significant digit of the input affects the least significant digit of the output, and the least significant two digits affect the least significant two digits, etc., we can generate possibilities recursively, digit by digit, checking as many digits as we have. This might look like it can blow up exponentially but a little bit of testing will show that it easily runs in time.

A slightly more efficient solution is possible if you note that carryless squaring of a reversed number is just the reversed result of carryless squaring of that number. This means that you can perform your recursion from the most significant digit to the least significant digit, simplifying the check for minimal correct result.

Checkerboard

The simplest solution to this problem is just four nested loops. The outermost loops over the A values, the counts up to individual a_is, the next loops over B values, and the next loops over individual b_is. The color of the output depends on the parity of the A index and B index, or, you can just keep track of the leftmost color of each row.

Coloring Contention

This problem looks challenging on first glance. We want to force Bob to make at least $k - 1$ color changes if the path length is at least k; this is the best we can do. And indeed, we can do exactly that, by coloring the edges by their distance from node 1. Initially we set all edges to uncolored. Edges adjacent to node 1 all get red. Then, edges adjacent to those edges that remain uncolored get blue. And so on, alternating colors, until all edges are colored. Essentially, the color of an edge is the parity of the distance from the source node. Edges that connect nodes at the same distance from the source node can get any color.

It is not necessary to actually color the graph; all you need do is calculate the distance between the two graph nodes and return that value less 1. Since the number of nodes is at least 2, you don't even need to worry about the case of a path length of 0.

Computer Cache

We'll start by ignoring the increment operation for now.

We'll build a segment tree with lazy propagation on the cache positions. In terms of the lazy propagation, any given node will be in either one of two states: either it will be completely empty, or we will save information of the form (data segment index, left endpoint of segment, right endpoint of segment). When we lazily propagate this information down towards the leaves, we can compute the boundaries for the new left and right endpoints. When we need to run a query, the left and right endpoint of the segment will match and we can reference the byte to print directly.

In order to handle the increment operation, we can build persistent segment trees with lazy propagation on the data segments themselves. When we now load data segments, instead of just tracking the data segment index, we also need to track how many increments had been applied to the segment at the time of the data load so we know which version

of the persistent segment tree to query for the correct data values. Since we only query for one byte at a time, if we maintain at each internal node how much each value in the given segment has been incremented by, we can walk the segment tree from root to leaf and sum all the increments, and add to that the original value to get our answer.

Alternatively, Fenwick or binary indexed trees can be used to solve the problem. To keep track of which data items and offsets are associated with each byte, one Fenwick tree can track the index of the command that set that particular range (in this case, the Fenwick tree accumulation function is max). In addition we have a Fenwick tree for each data item that tracks the increments in that data item (in this case, the Fenwick tree accumulation function is addition). You can relocate each query operation to right after last load operation for the query location, and reorder the results later to match the order the actual queries were performed in.

Dividing by Two

Since the only way to increase A is to add 1, if B is greater than A, you simply perform $B - A$ increments and you are done.

If B is less than A, however, you need to divide by two, and to do this you may need to add one first to make it even. So decreasing a number is performed by some number of increments interspersed with some number of divide by twos. How can we tell what is the optimal solution?

The key is the sequence $+, +, /$, or two increments followed by a divide by two. This is equivalent to $/, +$, which is shorter. So any time we have two increments followed by a divide, we can replace those three with the two-move shorter sequence.

So it's always optimal to do divides first, only using increments as needed to permit the divide to occur, until we get A to be less than or equal to B, after which we add as many increments as needed.

Error Correction

At first glance this is precisely maximum independent set on a graph; that is, find a maximum set of vertices in a graph that are not connected. Another way to look at it is to find the maximum clique on the complement of the graph, but the graph is sparse so it's probably best to work on the original graph. This is a well-known NP-hard problem, and the bounds clearly do not permit an exponential solution, so you should look for structure in the graph that can be exploited.

The structure we exploit in this case is permutation parity; the words are all anagrams, and thus are permutations of each other, and each permutation has a parity. Alternatively, you can look at it as the number of inversions on letters in each word, which also has a well-known parity property. This means that the graph is a bipartite graph, and maximum independent set can be calculated with any matching algorithm; the sum of the maximum matching and maximum independent set on a bipartite graph is just the number of vertices.

Any matching algorithm (Dinitz, Hungarian, or any network flow algorithm) can be used.

Even or Odd

The bounds on this problem are sufficiently small that just trying all possible starting numbers for a given N easily runs in time. But a little thought works too; for a given N, the only thing that matters is if the starting position is even or odd. Further simplification can be had by realizing that any group of four consecutive integers always sums to an even number, so you only need consider N mod 4. If $Nmod4$ is 0, the answer is always even. If $Nmod4$ is 2, the answer is always odd. If $Nmod4$ is 1 or 3, the answer is, it could be either.

Carny Magician

To solve this problem, we break it down into a number of easier problems, and build up solutions from there.

In order to construct the nth permutation that satisfies the requirements, we need to be able to count the number of ways to fill out a solution, given a particular prefix. We will use the case $n = 9$, $m = 4$, and $k = 3000$ as an example. We will write permutations as a simple digit string.

So for instance, in this case, we start on the left, and we try the prefix 1. This gives us a single fixed point, so the remaining 8 permutation elements must give us 3 more fixed points; this is equivalent to the original problem with $n = 8$ and $m = 3$. For these values there are only 2464 different permutations that work, so $k = 3000$ is too much, so we know the first digit is not 1. And, we can subtract the 2464 from the 3000 as we skip all the permutations that start with 1.

Next we consider the prefix 2, with $k = 536$. This is not equivalent to the original problem, as we are using a value that matches a slot number in the suffix. Thus, we actually need to solve a somewhat more general problem; we need to include the count of how many values are in the remaining set that could match one of the remaining slots. In this case, we still have the value 1 to assign, but slot 1 is used, so we have to solve the problem $n = 8$, $m = 3$, and $x = 7$ where the x indicates that we have only seven values that can match a slot, and the remaining one value cannot. So we define $g(n, m, x)$ as the number of permutations of n items with m fixed points where only x values can possibly match the slots numbers.

First, let's consider the different ways we can match up values. We want to match m of x values, so that's clearly $\binom{x}{m}$, so

$$g(n, m, x) = \binom{x}{m} g(n - m, 0, x - m)$$

From here on out we will always have $m = 0$. If $x = 0$, then there cannot be any fixed points, so $g(n, 0, 0) = n!$. Also, $g(0, 0, 0) = 1$.

Let's assume $x > 0$ and choose a location for one of the values x that might still match a slot. Either we can choose a location that still has a matching value but is not x (there are $x - 1$ of these), or we can choose a location that does not have a matching value (there are $n - x$ of these). In the first case, we use x but we also use up one of the matching slots, so x decreases by two; in the second case, x just decreases by one. So we end up with the recurrence

$$g(n, 0, x) = (x - 1)g(n - 1, 0, x - 2) + (n - x)g(n - 1, 0, x - 1)$$

From here the challenge is dealing with big numbers and overflow. If you use Python, you have no problems. Java BigInteger will work fine (but BigInteger is awkward to use). If you use C++, you probably need to write addition and multiplication routines that saturate at some value larger than 10^{18}.

Issuing Plates

The solution here is to create a list of the bad words. Then, when we read in a plate to check, we first substitute any digits for the letters they are similar to, according to the table. Next, we iterate over the bad words and simply see if the plate has one of the bad words as a substring, using the built-in library method for this purpose.

The most likely cause of a failure in this problem is not copying the table of digit to letter transformations carefully enough.

Glow, Little Pixel, Glow

There were two things you had to figure out to solve this problem. The first, was to determine when a horizontal and vertical pulse would actually intersect. The second was to figure out a way to sum the number of collisions without enumerating them.

For the first, just project one horizontal and one vertical pulse train onto the line $x = y$. The projection on that line has the pulse trains moving at exactly the same speeds, so if they overlap at any time they will always overlap; if they don't overlap, they never will.

For the second, build a list of events that occur, projected to the line $x = y$, of four types: a horizontal pulse start, a horizontal pulse end, a vertical pulse start, and a vertical pulse end. Sort them by their position on the $x = y$ line, and then scan from the first event to the last.

Every time we encounter a horizontal pulse start, we increment a counter of the currently active horizontal pulses; every time we encounter horizontal pulse end, we decrement that counter. Similarly for all the vertical pulses. In addition, for every horizontal pulse start, we know that pulse collides with all of the current vertical pulses, so we add the count of the current active horizontal pulses to our return value. For every vertical pulse start, we add the count of the current active vertical pulses to the return value.

Interstellar Travel

The first observation is that since the energy function is (piecewise) linear in the angle, the derivative is constant. This implies the best angle to launch the spacecraft will be directed at a star.

There are 10^5 stars, thus we cannot spend linear time to calculate the energy achieved from launching the spacecraft at a particular star. Instead, we must do a sweep, calculating the energy for all stars in one go. For each star, we can determine at which angle the star starts contributing to the energy of the launch, and similarly when that energy peaks and stops contributing. Again, since the derivative is constant, we can maintain the change in energy per radian moved, making it possible to find all energies in $O(n \log n)$ total time, after sorting by radian. Care must be taken to handle the circle properly, and stars that contribute energy no matter the angle must be special-cased.

Correcting Keats

We want to generate a list of all possible words at a Levenshtein distance of one from a given word, using letters from a given alphabet, without duplicates, and sorted. The easiest way to do this is to use a specific container that is sorted and unique (in C++, a `set`; in Java, a `TreeSet`). We simply iterate through the word position and make each possible change, adding the result to the container. At the end, we iterate through the container printing any words that are not identical to the given word.

Rather than iterating through positions, a particularly nice way to solve this problem is with recursion, keeping track of what character in the original string we are at, building an output string, and keeping track of whether we've made a needed change or not. See the file `Lev2_tgr.cpp` for this solution.

Maze Connect

We claim that this problem translates directly to a classic graph theory problem - given an undirected graph, compute the minimum number of edges to add to the graph such that the graph is connected. To solve this problem, we count the number of connected components in the graph. This can be done in a number of different ways - one way is to loop over all vertices and for every vertex which has not yet been seen, to run DFS or BFS and mark all vertices that are reachable from the current vertex. The answer is consequently one fewer than the number of connected components, as adding a single edge can reduce the number of connected components by at most one.

It remains to show how to translate the given problem into one that can be solved in the above manner. Imagine blowing up the input grid by a factor of 2 in each direction, and furthermore adding a boundary of empty cells around the grid. Every square in the grid represents a vertex, and two squares that share a side and are both empty share an edge. We can then run the above procedure on this graph.

Remorse

Intuitively, we want to calculate the frequency of each letter, and then use shorter codes for the most frequent letters. This intuition turns out to be correct, and can be shown by considering any case where this is violated, and realizing that the overall cost can be improved just by swapping the two codes.

Our first step is to calculate the frequency of each letter. This can be done with a simple array. Once we have the counts, we don't need the letter assignments anymore, so we can just directly sort the array.

Next, we need a list of codes, shortest first. We know we will need at most 26 codes, starting with a single dot, but we may not be sure how many codes we need to generate or how long they are. Again, we only need a count of code lengths, not the actual codes themselves. There are several ways to generate the codes; perhaps the easiest is to just generate, say, all codes up to 10 dots or dashes in length, and sort them; there are only 2046 of these so it will clearly run in time.

Another way to handle this is to just calculate how many codes of a given length there are, recursively. There is only one code of length 1, and two codes of length 3. If $f(n)$ is the number of codes of length n, then we can calculate this expression recursively with $f(n) = f(n-2) + f(n-4)$, with the first term giving codes starting with a dot and the second code giving codes starting with a dash. If you expand this out manually you'll easily see this is just the Fibonacci sequence, and the longest codes we need are of length 11.

After that, we match the most frequent letters with the shortest code and sum the length, adding three for each letter in the input except the first.

Perfect Flush

We sweep over the integers in the list in order and will attempt to construct the desired subsequence directly.

There are two cases to consider when considering the ith integer in the input list.

1. The given integer is already present in our subsequence. We do nothing.
2. The given integer is not present in our subsequence. This is the interesting case to consider.

   While our tentative subsequence is not empty, we will compare the given integer to the last integer in our subsequence. If the given integer is larger than the last integer in our subsequence, then append it to the end of our subsequence. Otherwise, it is smaller than the last integer. We can safely remove this integer if and only if there is another appearance of this integer that will be considered later in the sweep. We repeat this removal process until we can no longer remove an integer, at which we perform the append.

Radio Prize

You are given a tree with N nodes, where every node has a tax value t_u and each edge has some weight w_i. The cost of a path between nodes u and v is equal to $(t_u + t_v)\mathrm{d}ist(u, v)$. For each node u, compute the sum of the costs of all paths to all other nodes v.

The above expression can be broken up into $t_u\mathrm{d}ist(u, v) + t_v\mathrm{d}ist(u, v)$. Compute for some arbitrary root in $O(N)$ time. Then computing the answer for a neighboring node can be done in $O(1)$ time.

Fix some node u as the root, compute two quantities:

$$a_u = \sum_v \mathrm{d}ist(u, v)$$

$$b_u = \sum_v t_v\mathrm{d}ist(u, v)$$

Then the answer for node u is just $t_u a_u + b_u$.

How do we compute $a_{u'}$ and $b_{u'}$ for some neighbor u' of u?

Let w be the length of the edge between u and u'. For all nodes in the subtree of u' when the tree is rooted at u, their distance to the root decreases by w. For all other nodes, the distance increases by w. If we let $size(u)$ be the size of the subtree rooted at u, then

$$a_{u'} = a_u + w(N - size(u)) - wsize(u)$$

Similarly, if we let $tax(u)$ be the sum of the tax values of all nodes in the subtree rooted at u, then

$$b_{u'} = b_u + w(tax(\text{root}) - tax(u)) - wtax(u)$$

Since these values can be updated in $O(1)$, walking and updating the tree and computing all values takes $O(N)$ time in total.

Rainbow Strings

For each lowercase letter l, let $f(l)$ be the number of times that letter appears in the string. We claim the answer is

$$\prod_l (f(l) + 1).$$

The combinatorial interpretation for this is as follows - for each rainbow string, either the given letter appears in the string or it does not. If it appears in the string, there are $f(l)$ choices for which index to select. If it does not appear in the string, there is 1 way to make that happen. This means there are $f(l) + 1$ valid choices for which index to select for the given letter. All letters are independent, so we multiply $f(l) + 1$ over all letters.

Speeding

For simplicity, we'll compute the maximum speed we can be certain the car drove. To compute the greatest integral speed, we take the floor of the answer.

Let's first focus on the case where $N = 2$. In this case, we know that the average speed the car was driving at was $\frac{d_2 - d_1}{t_2 - t_1}$. The maximum speed can therefore be no larger than that, as the car can just drive at that speed without no changes.

We will use the solution where $N = 2$ to solve the case where $N > 2$. In particular, we assert that the answer is simply $\max_{1 \le i \le N-1} \frac{d_{i+1} - d_i}{t_{i+1} - t_i}$. The reason for this is that any two consecutive photos can be treated as a case where $N = 2$, which provides a tight upper bound on the speed that the car could have been driving over that segment of road. We can therefore just take the segment of road with the highest upper bound.

Pivoting Points

Like many geometry problems, we tackle this one by focusing on the discrete events that shape the continuous motion of the windmill. In this case, the event that we care about is the promotion event, where the line hits a second point

and switches pivots. We can see that there are $O(n^2)$ distinct events, as they are fully parameterized by the point that is currently the pivot, the point that is about to be promoted, and the orientation of the line with respect to the points (the line is oriented because we are tracing its path through a 360 degree rotation). The problem statement provides us with a hint that the line will always return to the same position after a 360 degree rotation. This is useful because it means that the $O(n^2)$ events can be partitioned into disjoint cycles, and each possible windmill is simply a traversal of one of these cycles at a different starting point. Therefore if we simulate the motion of a single windmill by traversing its events, we can compute the promotion count for all the points during not only that windmill, but all other windmills that share its cycle. We also observe that there are $O(n)$ different cycles, since every windmill must at some point be oriented straight up and must pass through at least one of the n points while doing so.

Therefore the bulk of the work will be in simulating the motion, which requires finding the next event that will occur after the current event. Naively, we could perform the promotion of the current event and then check all of the other points to see which point will get hit earliest as we continue rotating the line about the new pivot; this would be $O(n)$ work per event, for a total of $O(n^3)$ work. This is not quite fast enough for the given bounds. To help speed up this search for the next event, for each point we maintain the order of all the other points sorted by angle; this allows us to either step or binary search to find the next event. This requires $O(n^2 \lg n)$ precomputation, followed by $O(\lg n)$ work per event, for a total of $O(n^2 \lg n)$ work.

Note that the events look a little different depending on whether the next point hits the line above the current point or below it. Mirroring points about the pivot in the right way can make the computations cleaner.

This page is intentionally left blank.