# Problem A. Anton's ABCD

Unfortunately this problem seems to be an unintentional variation on https://atcoder.jp/contests/agc055/tasks/agc055_b. I meant to create a problem in the style of Anton, not make a problem about exactly the same setup, oh well.

In our setup, if there's some substring, that is a rotation of ABCD it can always be moved one spot to the left. For convenience let's call such substrings chunks. If the previous character is X, we rotate the chunk until XABC becomes a rotation itself. By repeating this, we form a string of the form: ABCDABCDABCD...ABCD garbage. Where garbage does not contain any chunks. It turns out in whichever order substrings are moved to the front, the number of repetitions of ABCD at the front is the same, and the garbage also is the same. For a proof of this, see the AGC problem.

Another way to phrase it, is that each time you remove some chunk, until you are left with only garbage. This perspective is handy for counting, because it shows that in a string it can always be divided into recursive structures of rotations of ABCD, with possibly more rotations inside them, interleaved with the characters from the garbage string.

So let us first define a block. A block can be defined recursively as block = W (block) X (block) Y (block) Z (block), where WXYZ is some chunk. The brackets indicate that these positions possibly contain a block or are empty. For the inner (blocks), their starting character is restricted to only 3 choices, because otherwise we would count the same string multiple times with different block structures.

We can calculate the number of different blocks of length $4k$, for some $k$ with dynamic programming. block$[k]$ = number of distinct blocks and helper dp:

combiDP$[num][k]$ = number of ways to pick $num$ blocks of total size $4k$

For calculating block$[k]$ we can pick which (blocks) are empty, and for all nonempty blocks do times $\frac{3}{4}$, and look in the corresponding combiDP for finding out the number of ways to choose $0 \leq v \leq 4$ blocks with total size $4(k-1)$.

The total string will be an interleaving of the characters of garbage and multiple blocks.

So these can be calculated with a dynamic programming with state:

words$[n][k]$ = number of words where there are already placed $n$ garbage characters and blocks having total size $4k$
The transitions of this are straightforward, putting a block of some size + a garbage character, or a garbage character after the current word. The only tricky thing left is when in these transitions to do $\frac{3}{4}$ or not. This comes down to if there have already been placed garbage characters or not. From this DP it becomes evident that the actual value of the garbage characters themselves does not matter, so the answer only depends on the length of the word and the number of chunks.

The total time complexity is $O(nk^2)$, where $k$ is the amount of chunks in the string. This might seem slow, but because $k \leq \frac{n}{4}$ it is fast enough.

# Problem B. Beer Circuits

Because we are minimizing a maximum, a first logical thing to do is binary search on the answer. If we want to check if the answer is $\leq M$, then we can draw an edge between any two nodes which are at distance $\leq M$. We want to check if in this graph there is any simple cycle with length $\leq k$. There are two problems, the graph could be pretty dense, and it also seems hard to obtain the graph anyway.

The first concern is not an issue. Whenever some node has degree 6 or more, we can observe the following: There are two of its neighbours which have an angle between them of less than $360/6 = 60$ degrees. Because both are distance $\leq M$ away from the degree 6 node, it means those neighbours are close to eachother, and also have an edge. These edges form a triangle, and this is always a valid beer circuit. So we do not have to find more than $2.5n$ edges, because otherwise it always leads to some degree 6 vertex by the pigeonhole principle.

How to build the graph fast? Here we can make use of a grid. If we make a grid of cells of size $M \times M$,

then the only candidate neighbours for a point are the ones in neighbouring cells (8-adjacency) and its own cell. It turns out we can actually bruteforce over all nodes in these cells and check if a pair of nodes actually has an edge or not. This may seem like it could degenerate to $O(n^2)$, but actually in each cell, not more than $O(1)$ nodes can be placed, before some node in that cell gets a degree of 6, and if we stop the algorithm each time a node gets a degree of 6 or higher, we never have to check more than $O(n)$ candidate edges.

To implement this, it's tempting to use a hashmap. But this turns out to be kind of slow, due to its quite big constant and also the hidden constants in the $O(n)$ candidate edges we have to check. A better way is for each point $p$, calculating $q = (\lfloor \frac{p.x}{M} \rfloor, \lfloor \frac{p.y}{M} \rfloor)$, sorting these points, and then doing a couple of passes of 2 pointers, to find all candidates. To make this even faster, you could use radix sort, for sorting the $q_i$'s, but this was not needed.

Using this, we can at least find in $O(nlog(C^2)) = O(n \log(C))$, where $C$ is the bound on the coordinates, with quite a big constant. An extra log factor only for sorting a list of $n$ pairs, is also not too bad, so long as there are no log factors in the tight loop.

But this only gives us an upperbound on $M$, because there could still be slightly longer cycles in the graph, that have all edge lengths $< M$.

What would happen if we run the binary search iteration for the exact $M$ such that after this triangles appear? And what if we do not early break from the construction algorithm? It turns out it still runs in $O(n)$ time and the full graph it produces has only $O(n)$ edges. To see this, note that for all pairs of points with distance $< M$, they will have no degree 6 vertices, because otherwise $M$ would be smaller, so those are at most $2.5n$ edges. All the edges of length exactly $M$ are also bounded by $O(n)$, and to see this we can draw the grid of cells of size $M \times M$ again, and due the graph of all edges $< M$ not containing a triangle, each cell in the grid has $O(1)$ points in it, and the number of edges of length $M$ connecting to some point must be also $O(1)$, because those edges can only form between neighbouring cells or two points in the same cell.

So now we have a sparse graph with $O(n)$ edges, with weights equal to the distance, each node has degree $O(1)$, and we want to find cycles of length $\leq k$ in it, such that the maximum edge weight in the cycle is minimum.

To solve this final part, we start with an empty graph, add the edges in increasing order of weight, and when adding an edge, doing a BFS in the current graph, from node $u$ to node $v$. If we bound the max distance in our BFS by $k$, and break the BFS after this, it will only visit $O(k^2)$ nodes. (Proof left as an exercise to the reader, it is also based on the sparsity of the points in the $M \times M$ cell grid.) Because each node has degree $O(1)$, it means each BFS runs in $O(k^2)$.

In this process we need to keep the best maximum edge in the cycle so far, the number of edges in the best cycle of minimum length, and the number of ways updated. For counting the number of cycles going through edge $uv$, we can do DP on the shortest path DAG which is made with the BFS.

In total, this part of the solution uses $O(n(\log(n) + k^2))$ time.

# Problem C. Curly Palindromes

This problem may look hard (strings + geometry??), but can be solved with simple casework.

- If all points have a distinct label
  - The answer is 1
  - Consider the middle of the palindrome
  - Either the palindrome is even-sized, then the middle letters need to be the same, but this is impossible.
  - Or it is odd-sized, and then you get problems with making an angle of exactly 180 degrees, with two of the same letter right next to the middle $\ldots ABCBA \ldots$

- Else if all points lie on a line:
  - The answer cannot be bigger than 2 due to the curly constraint,
  - A palindrome of length 2 is easily made.
- Else:
  - Take any two points with the same label $A_1$ and $A_2$.
  - There must be a point not on the same line, call this point $B$, its label does not matter.
  - This can give you unbounded curly palindromes, of $\ldots A_2BA_1BA_2BA_1\ldots$

# Problem D. Division 3 Polyglot

For $x \leq 24$, this test works:

$$1$$

$$2x+1$$

$$x\ 1\ 1\ 1\ 1\ 1\ \ldots 1$$

For $x = 25$, it is a square number, so this test works:

$$1$$

$$50$$

$$5\ 5\ 5\ \ldots\ 5$$

# Problem E. Enumerating Substrings

If we could somehow guarantee that strings can never overlap, $F(S,P)$ would be just the number of occurrences of $P$ in $S$. Unfortunately there can be overlaps. If two substrings $P$ overlap, $P$ can be written as $wxw$, where $w, x$ are two strings. The overlap would look like $wxwxw$. Because the patterns should be beautiful, it turns out that for a fixed pattern $P$, there can be only one $w$. Proof: If there are $w \neq w'$, such that $P = wxw = w'yw'$, then if we look at the first character, it must be repeated at at least two other places. This contradicts the beautifulness of $P$. It is also the case that borders of length $> |P|/2$ are not allowed, because these borders would give rise to a smaller border, which is again a contradiction.

For the counting, we can first consider all possible lengths $|w|$, count the $F(S,P)$ over all $S$ and $P$, with $P = wxw$. Finally we also calculate $F(S,P)$ over all $P$ which have no border. This last category of $P$'s can be easily be counted by subtracting all words with a border from the total number of beautiful words.

This splits the problem up in the subproblems:

- Calculate the number of beautiful words of a certain length $n$
- Calculate the number of beautiful words of the form $wxw$
- Figure out a way to accurately count occurrences of $wxwxwxw$ in $S$ and how to deal with overlaps.

First we can use a straightforward $O(m^2)$ dynamic programming to calculate: beautiful[length][distinct characters] = how many possible strings

For calculating the number of ways to make $wxw$ for a fixed size $|w|$, we bruteforce the number of distinct characters in $x$, and sum up beautiful$[|x|][d] \cdot \frac{(k-d)!}{(k-d-|w|)!}$. In this formula we make use of the fact that the characters that $w$ and $x$ use must be disjoint, because otherwise the string would have a character appearing $\geq 3$ times, and $w$ must consist of all distinct characters.

Now the only thing left is understanding how to count overlaps correctly. Given a fixed $|w|$, it turns out to be good enough to be able to count the number of occurrences of $wxw, wxwxw, wxwxwxw, \ldots$ in all $S$. For each of these, there are $n - \text{length} + 1$ positions to place the string, and $k^{n-\text{length}}$ ways to choose the remaining characters in $S$. So these can be calculated in $O(1)$ each, with some precomputation of powers of $k$. These, multiplied by some coefficients can be made equal to $F(S, P)$, the coefficients can be found using a inclusion-exclusion argument, where you look at maximal substrings of the form $wxwxwxwxw$ (maximal meaning they cannot be extended further with $xw$ at the end or $wx$ at the front). You count the number of occurrences of each string of the form $wxw, wxwxw, wxwxwxw, \ldots$ in this substring. The number of occurrences times some coefficients, should be equal to $\lceil \frac{k}{2} \rceil$, where $k$ is the number of $x$'s in the maximal word.

The following choice of coefficients works: For fixed |w|, the following sum correctly calculates the $F(S, P)$ for some fixed pattern $P$:

$(\#wxw) - (\#wxwxw) + (\#wxwxwxw) - \ldots.$

What does the time complexity become if we naively evaluate the sum for every $|w|$? It will be $O(\frac{n}{m}m) = O(n)$, so this is fine.

The total time complexity becomes $O(n + m^2)$

# Problem F. Football in Osijek

For some valid team, it must be a directed path in the graph consisting of the edges $i \to a'_i$, such that it cannot be extended any longer, by appending more nodes to the back. Here $a'_i$ are the final preferences after all operations.

So the goal is to create a maximal directed path of a specified length. The number of operations we need to do for some path is equal to the number of edges in our proposed path that are not in the original graph, as we have to change them. There's one extra difficulty, in that if we look at the last person $v$ in the directed path, its $a_v$, must be already in the path, or we need to do one more operation to change this $a_v$, because our path is not maximal.

Firstly we want to handle the case where doing no operations already gives a valid path. The graph is a functional graph, so it is a collection of cycles with trees hanging off them. For one of the components of this graph it can make all sizes of paths in the interval [cycle size, cycle size + maximum tree height ]. So we can calculate all these intervals, find all $k$ on which answer 0 is achievable in linear time.

For all other cases, we must find the minimum number of disjoint directed paths in the original graph with total size $\geq k$. Clearly this number minus one is a lowerbound on the number of operations. Actually if this lowerbound is not equal to 0, it is also always achievable. And because we already have a special case for when the answer is 0, this gives a framework to find the optimal solution for all $k$.

The proof that the lowerbound is achievable comes from the algorithm to find the directed paths.

To solve the remaining subproblem, we can first model the problem as a minimum cost flow problem. Except for the functional graph, add a source and a sink. Connect every node to the source and sink by a directed edge. Make all edges unit capacity, and all functional graph edges cost $-1$ and source edges cost $-1$, while the sink edges have cost 0. Also make the nodes have capacity 1, by the simple transformation of cutting each node in 2 and connecting in- and out-edges to the two opposing sides. The minimum-cost $v$-flow in this graph corresponds to the maximum length of disjoint paths you can make using $v$ paths.

Finding the mincost $v$-flow for all $v$, gives us a polynomial time solution.

To optimize it, we optimize the generic successive shortest paths algorithm for this specific graph.

We need to find a shortest path in the residual graph. Normally this is complicated because the residual graph also contains reverse edges to undo flow. But it turns out that you never have to take a reverse edge to undo flow. The only edges that you can undo are the functional graph edges and the edges that split a node into two, and these now have cost 1 and 0. But making a path through these only increases the cost, and the only way to the sink is to either take another cost 1 or 0 undo edge, or a direct edge to

the sink. So it only makes costs for going to the sink worse.

So greedily taking the largest available directed path, and never undoing anything is optimal. To implement this efficiently we can calculate the subtree heights, and make a longest path decomposition of the trees. Cycles are also easily dealt with by replacnig it with a big node with the trees hanging off it, making this node have a weight of cycle size, instead of 1. The total complexity is $O(n)$.

From this algorithm it is clear, the first path we choose always uses one of these big nodes. So it has the property that its last edge points back to inside the path, so in the $v$ paths that are chosen, we can always use this one as the last of the $v$ paths, connect all the other paths in a chain using $v-1$ operations, and shortening the last path slightly, to get exactly $k$ nodes in the final path.

# Problem G. Graffiti

The crux of this problem is the $|w| \leq 3$ constraint. If we look at all cases of $|w| < 3$, they turn out to be trivial.

$|w| = 1, w =$ `a`: All the same letter, $n$ paths.

$|w| = 2, w =$ `aa`: All the same letter, $2(n-1)$ directed paths.

$|w| = 2, w =$ `ab`: 2-colour the tree. This way each edge is a directed path in one direction, and this is the best you can do anyway. $n-1$ directed paths.

Actually one more trivial case is $|w| = 3, w =$ `aaa`, but we can handle that with minimal extra work with the solution for the other cases as well.

The main approach for $|w| = 3$ is to make a smart dynamic programming. We can make our state `dp[subtree of u][letter on myself][letter on parent] = maximum number of good directed paths possible, in the subtree of u, plus the parent node`

If somehow the child dp states are merged into a subtree DP state for $u$, the only paths that are missing are the paths, that have their centre at node $u$. These paths could come from the parent and go to a child, vice versa, or between two children. Now we have to look at all cases of possible words, and check what new paths get created.

For $w =$ `aaa`, the number of new paths created is $k(k-1)$, where $k$ is the number of letters `a` placed among all children and the parent of $u$.

For $w =$ `aba`, the number of new paths created is $k(k-1)$, the same.

For $w =$ `abb`, the number of new paths created is $k \times (\deg(u) - k)$, where $k$ denotes the number of `a`'s.

For $w =$ `abc`, the number of new paths created is $\lfloor \frac{k}{2} \rfloor \times \lceil \frac{k}{2} \rceil$, where $k$ is the number of non-`b`s. This is the only tricky formula, and it basically comes down to the fact, that for some node, the best we could do is if the number of `a`'s and `c`'s are as equal as possible. We can always achieve this, because in a subtree, we can flip all `a`'s and `c`'s, keeping the same score for this subtree, but changing the balance in the current node.

Notice that in each formula, it only depends on one variable $k$, which may mean something different, but the nice thing is, it is bounded by the degree of the node. So we can actually brute force $k$ for a node $u$, and try to find the optimal solution for a fixed $k$. Furthermore, in each case, we really have only two different kinds of letters. (letters `a` and `c` can be grouped together, because it doesn't matter anyway which one it actually was.) So we can conclude that in our letter state in the DP, only 2 different states are needed letter that does increase $k$ or not.

Now it comes down to taking some max convolution of the DP states of the children, getting an array of $best_k =$ `best sum of children DP's states, with this particular` $k$, and afterwards adding $f(k)$ to each entry, our formula based on the case.

This is easily optimized to $\deg(u)\log(\deg(u))$, by noticing that for this max convolution, it is optimal to take the $k$ children that do $+1$ `to` $k$ to be the children with maximum `dp[c][+1 to k][parent] - dp[c][+0 to k][parent]`, and sorting with respect to this.

This solves the problem in $O(n \log(n))$ and we don't have to implement DP 3 times either, we can just pass a lambda function with our formula based on $k$ instead.

# Problem H. Huge Oil Platform

First some observations about the optimal solution. If we fix the set of locations that oil is extracted from, it basically turns into the minimum perimeter covering rectangle problem. It is well-known that the optimal covering rectangle must have at least two points on one of its sides. Or at least, I thought this was well-known until I wanted to search for a proof. Couldn't really find it, so instead I made my own proof.

## 0.1 Proof

There are some edge cases. If there's only 1 point, the optimal "rectangle"has perimeter 0, and degenerates into a point.

If all the points lie on a line, the optimal rectangle is obviously the degenerate rectangle which is the segment connecting the farthest points. Note that although this is an edgecase, it still shows that two points are on one of the sides of the rectangle.

Then for the general case: Firstly we only have to cover the convex hull of the point set we want to cover. If we look at the optimal placement of the rectangle, it must have at least one point of the convex hull on each of its sides. If this optimal solution has already 2 points on one of the sides, we are done. otherwise each edge is constrained by exactly 1 point. If we now look at slightly rotating the rectangle, while keeping these 4 points on the sides, we can look at the perimeter as a function of the angle. With some trigonometry, we get the formula

$P(\alpha) = 2||a - c|| \cos(\alpha) + 2||d - b|| \cos(\alpha + \beta)$, where $a, b, c, d$ are the 4 points on the rectangle, labeled in circular order, and $\beta$ is some angular offset based on the placement of the points. Some of the points could be the same point, this doesn't matter for this proof. This perimeter function is only correct on the interval of angles that these 4 points are the extreme points in their corresponding directions. On the edges of this interval, it means that two points are both an extreme point, so two points are both on one of the sides. Note now, that on this valid interval, the function is the non-negative linear combination of two concave functions (cos is not concave everywhere, but our interval is only restricted to a part where both cos functions are positive). This means the minimum of these functions has to be in one of the edges of the interval ■

With these observations, a brute force solution is possible. Take all pairs of points as the pair of points that lie on one of the sides. This fixes the rotation of the rectangle. Then 3 more points have to be chosen to constrain the other three sides of the rectangle. Afterwards, check which points lie in this candidate rectangle, and this gives a $O(n^6)$ solution.

To optimize it, we can keep the brute force solution, but optimize the inner loops. So we still brute force all pairs to determine the placement of the first side. Now we sort all the points on the vector perpendicular to this side, and try all possible rectangles in increasing order of height. For a fixed height the problem actually boils down to a biggest subarray sum problem. Where you have some points with $x_i$ coordinates (this coordinate axis is pointed parallel to the first side) and a weight $w_i$. We want to maximize $\sum w_i - 2 \max x_i + 2 \min x_i$ (this counts the profits minus two times the length of the interval, because of the perimeter.)

Because we do sweepline, we update the set of points in this subarray problem a bunch of times. To do this fast, we can use a segment tree that's very similar to the max subarray sum segment tree. The time complexity becomes $O(n^3 \log(n))$, where the $\log(n)$ is due to sorting and the segment tree operations.

Let's prove that this algorithm is precise enough when implemented using doubles: Inside the segment tree, there are floating point values added and subtracted, which could make the relative error pretty big, due to catastrophic cancellation. Let's bound the largest value present in the segment tree: Each point has new coordinates in the new coordinate system, those are of order $O(A)$, where $A$ is the maximum

coordinate in the input. If we let the maximum profit in the input be $P$, then the sum of profits, inside the segment tree is of order $O(nP)$. This means that the values in the segment tree are bounded by $O(nP + A)$, and any value in the segment tree is calculated as a sum of $O(n)$ values, so the maximum absolute precision error in any of the segment tree nodes is $O(n(nP + A)\varepsilon)$. We know that we can make a degenerate rectangle which only covers the maximum profit location in the input, so ans $\geq P$. So the relative error on the answer is $O(n(n + \frac{A}{P})\varepsilon)$. With some more work you can show even better error estimates. But as $n \leq 400, P \geq 1, A \leq 10^6$ and for doubles $\varepsilon \leq 10^{-15}$, this analysis shows it to be precise enough.

There's also an alternate solution which has a better constant factor and better error guarantees. Instead of solving a subarray sum-esque problem with updates, we can instead use the information that the two points that we bruteforced, should be on the edge of the rectangle. This makes it such that we can solve two independent problems to the left and to the right of these two points separately. This simplifies the operations on the nodes in the segment tree, and gives better error guarantees, intuitively because we are not subtracting very big coordinates.

# Problem I. Isomorphic Delight

Let's get some special cases out of the way.

- For $n = 1$ it is possible, and the graph is the graph with one node.
- For $2 \leq n \leq 5$ it is impossible, which can be proven with a bruteforce, but it is also not too hard to do on paper.
- For $n = 6$, this is a very special case, because here there's exactly one optimal graph (up to isomorphism) that consists of a cycle of length 3, with paths of length 1 and 2 hanging of off it.
- For $n \geq 7$, this is the tricky, which can be handled with exchange arguments.

Say we have a component which has at least one cycle. If it is size $k$, it must have $\geq k$ edges. Now, if there are multiple components, we can do the following transformation:

Delete the cycle component and the biggest remaining component. Using these nodes, build a big tree, that is asymmetric. Because a graph with cycles has size $> 1$, and because we assumed we had an asymmetric graph to begin with, the cycle component has size $\geq 6$, so it together with another component has size $\geq 7$, and there exist asymmetric trees for each size from 7 onwards. If you look closely, this transformation preserves or improves the number of edges, while it decreases the number of non-tree components. This exchange argument also proves that there are no optimal solutions where some component has more than 1 cycle.

This fact is used in the checker, to avoid having to check the asymmetric-ness of complicated graphs. Actually, if there's some component with more than 1 cycle, while the solution still achieves the least number of edges, it must mean there's a contradiction in one of the earlier tree/functional graph components.

So actually, for an optimal solution, we want to greedily pick the $k$ smallest asymmetric trees, that are pairwise not isomorphic, such that they together have $n$ nodes. If adding the $k + 1$-th tree would use up too many nodes, but the smallest $k$ have too little nodes, we can instead increase the size of the biggest tree, towards $n -$ size of other trees. One possible construction for this is a tree with a "root"node and paths of length $1, 2$ and $n - 4$ hanging off it.

For enumeration of small asymmetric trees, we first start with trying to enumerate asymmetric rooted trees. (So trees with a root node which has a different colour from all the other nodes) This can be done by inductively making all the asymmetric rooted trees from smaller sizes to bigger sizes. For making an assymmetric tree of size $n$, we take some set of trees as children subtrees of total size $n - 1$ and add a root, connecting to these subtrees. This can be implemented by storing all asymmetric trees of all sizes, and with backtracking finding all sets of total size $n - 1$, and combining these sets into a bigger tree. If implemented correctly, this will take only time linear in the output size.

it turns out that the enumeration of unrooted trees is not that much harder. Let's look at the centroid of some asymmetric tree.

- If it's unique, we can take this as root node, and our tree is almost an asymmetric rooted tree. The only difference is that the subtree sizes of its children subtrees have to be $< n/2$. These can be enumerated with exactly the same backtracking approach.
- If it's non unique, there are exactly two neighbouring centroids. The edge connecting these centroids partitions the tree into two rooted asymmetric trees of size $n/2$. The only catch is that these two rooted asymmetric trees cannot be isomorphic, otherwise the whole tree would have a symmetry. Luckily we could already enumerate rooted trees, so this also makes this case easy.

For truly linear time, we would have to prune the backtracking to quit immediately whenever there is no way to extend the partial set of trees to a full set of trees of the necessary size. But since usually the partial solution can be completed, and there are no big useless branches this is not needed.

# Problem J. Jumbled Primes

If there was no query limit, we can first make sure that we can actually distinguish primes from nonprimes. Let's say a number is composite. Then there are two cases. Either it is a multiple of two distinct primes. If we find that for a specific number the set of gcd's with all the other numbers contains at least two different numbers $> 1$, it's definitely composite, and with this check we can filter out all these composite numbers.

The other case is that the number is a prime power $p^k$, with $k > 1$. If $k > 2$, then among its *gcd*'s it has at least $1,\ p,\ p^2$ so we can distinguish those. For $k = 2$ there could potentially be trouble. Luckily we then now that $p \in \{2, 3, 5, 7\}$, and for each $p^2$, out of this group there at least 2 numbers that are divisible by $p^2$. For example $p = 7$, gives 49 and 98. So for each of these $p^2$, its *gcd*'s contain $1, p, p^2$. So we have solved the problem in $\binom{n}{2} = 4950$ queries.

Now the problem becomes about efficiently using queries. Because we only care about the average number of queries over a lot of testcases, we want to optimize the expected number of queries, and don't care about the variance. There are lots of potential optimizations. While testing, errorgorn got the number of queries down to 500. Here only the most important optimizations are listed.

First, notice that there are quite few primes less than $\sqrt{n}$, only $2, 3, 5$ and $7$, let's call these the small primes, and the other primes $\leq n$ the big primes.

Our basic plan will be:

1. Find multiples of $2, 3, 5$ and $7$
2. Trial divide all the numbers to know their divisibility by the small primes, by using gcd queries.
3. For all numbers divisible by only one small prime, find the prime powers, and filter them.
4. For all numbers that are left in some group of numbers divisible by a small prime, filter out all the numbers of the form $p \times P$, where $P$ is a big prime, then we finally know which number in this group is the actual small prime $p$

## 0.2 Finding Multiples

For finding multiples of small primes, we can use randomness. The expected time until a random pair we query has both numbers divisible by $p$, is about $p^2$. We are waiting until we have at least one pair for all small primes. We can roughly upperbound this expected number of queries by $2^2 + 3^2 + 5^2 + 7^2 = 87$, but in reality it is less.

To make the next trial dividing phase more efficient, let us find multiples of $5, 6$ and $7$ instead. This does not use that many more queries, but it saves a lot of queries, because the $\gcd(6k, x)$ query gives us info about 2 and 3.

### 0.3 Trial Division

If we query the numbers in order, 6, 5 and then 7, and we stop whenever a number is divisible by more than 1 distinct prime, it can be experimentally verified that this phase is quite efficient.

Next up the numbers are divided into a few groups:

Known composites, Big primes $\cup \{1\}, \{2k \text{ for some } k\}, \{3k\}, \{5k\}, \{7k\}$

Numbers in the second group must be the number 1 and all the big primes, because they don't have any factors $\leq \sqrt{n}$.

### 0.4 Finding Prime Powers

In a group consisting of numbers $pk$, for some $k$, it will only contain numbers of the form $p^{e_1}$ or $p^{e_2} \times P$, where $P$ is some big prime, because all other numbers were already filtered, during the trial division.

Again, we make use of randomness, to get good expected number of queries. Our objective is to find two numbers which are both divisible by $p^2$. Because we know all numbers are already divisible by $p$, this will take about $p^2$ queries once again. There are a few edgecases for $p = 5$ and $p = 7$. To even have a chance of getting two numbers divisible by $p^2$, we must actually in this random querying also use numbers that we already determined that were composite, so not only numbers out of the group.

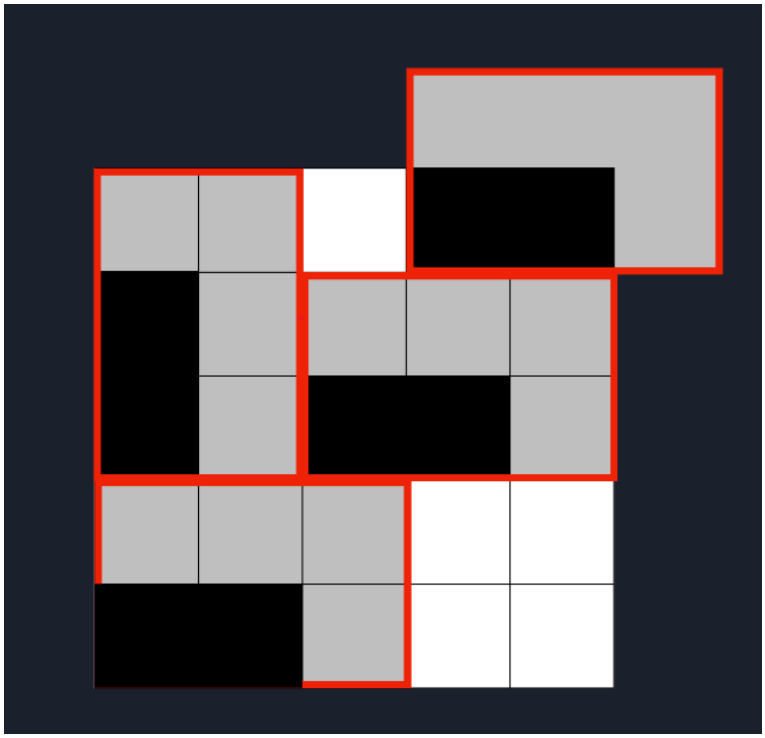
### 0.5 Trial Division 2

Finally in the group of numbers that we have left, they are only $p$ or $p \cdot P$ for some big prime $P$. By trial dividing all numbers in these groups by the big primes until we find a number which is not divisible by any big prime, we get the small primes. For extra optimization, we can go in order of $2, 3, 5, 7$, we can filter out big primes that are already too big in the previous iteration, and did not divide any number in the set, which saves us queries in the later iterations of this process.

When implemented carefully this gets about 560 queries on average. A nice optimization which does not require much work, is to add a cache to not ask queries multiple times between the phases.

## Problem K. King's Dinner

There is a bijection between filling an $n \times n$ grid by placing dominoes without touching, and filling an $n+1 \times n+1$ grid using rectangles of $2 \times 3$ and $3 \times 2$, that may touch but may not overlap. The bijection is putting dominoes of the same type in the topleft of the rectangles and vice versa:



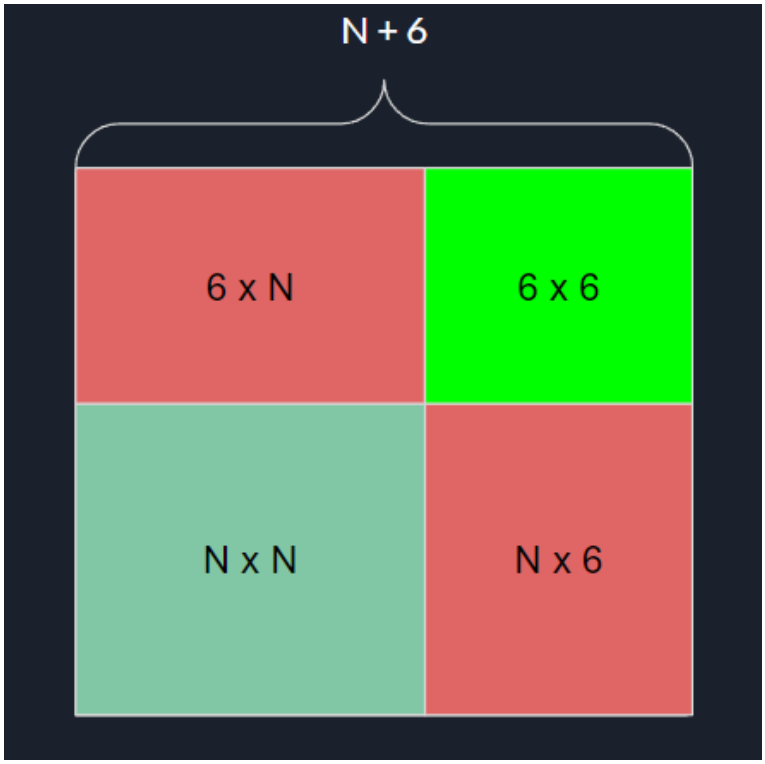
Bijection

This bijection works because the empty squares around a domino are mapped to exactly one rectangle. To verify it completely, all cases where dominoes are closeby can be inspected.

In the perspective of placing the rectangles and bounding the maximum area of the number of rectangles, we see that we cannot place more than $\lfloor \frac{(n+1)^2}{6} \rfloor$ rectangles.

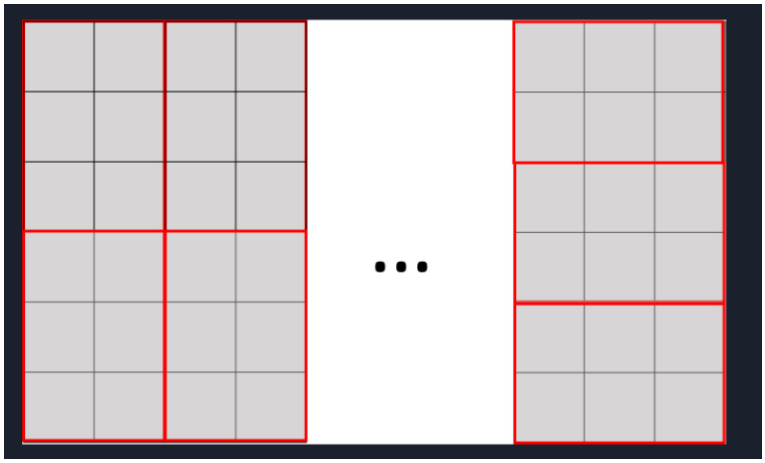
This number is always achievable. For $n = 1 \ldots 7$ this can be checked by hand, and is left as an exercise to the reader.

Next up we can make an inductive step, that given an optimal grid for $n \times n$, transforms to a grid of size $n+6$, covering all natural numbers. The easiest way to explain it is in the rectangle perspective. First we make an optimal grid of $n \times n$, then append pieces of $6 \times n$, $n \times 6$ and $6 \times 6$ to it:




Induction step

$6 \times 6$ we already know how to tile perfectly with rectangles. $6 \times n$ we can tile using two rectangles stacked on top, with possibly three rectangles on top of each other, if $n$ is odd:



Construction for $6 \times n$

This finishes the proof, and the algorithm is just implementing these steps recursively. The time complexity is $O(n^2)$ or worse, but since $n \leq 100$, time complexity is not a big issue.