Baby Bob's Coloring Book (circlecoloring)

Author: Zsolt Németh

Developer: Zsolt Németh

Solution

The first observation is to notice that if there is a cycle consisting of an odd number of circles (which will be referred to as vertices from now on), then there will be at least one pair having the same color in that cycle. On the contrary, even cycles can be colored (e.g., using alternating colors) such that no adjacent pairs are having the same color. This is enough to solve the first subtask.

The second important observation is to note that the new edges added by Baby Bob can be interpreted as breaking a cycle into two smaller cycles that share exactly one common edge (the one drawn by Baby Bob). We must also note that each cycle will have at least one edge that is not a common edge with any other cycle – the condition that every vertex is connected to at most one other vertex by a new edge guarantees this.

If we just consider the original cycle and $M = 1$ new line, that line can break the original cycle into two cycles such that the number of vertices is either odd-odd, odd-even, or even-even.

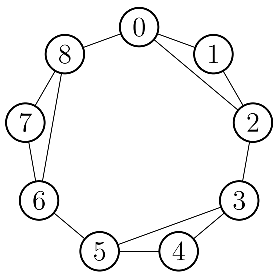
- In the first case, we can color the endpoints of the common edge using the same color and every other vertex using alternating colors to obtain the minimum number of pairs (=one pair) with the same color.
- In the second case, color the endpoints of one non-common edge with the same color and everything else using alternating colors.
- In the third case, we can color everything alternatingly to have 0 pairs with the same color.

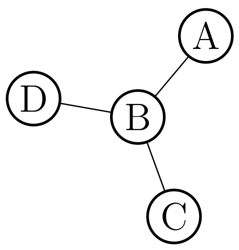
This solves the third subtask.

We know that each even cycle can have zero pairs with the same color and each odd cycle must have at least one. It is never worth coloring adjacent vertices of an even cycle using the same color, as we can just recolor the cycle and if a shared edge had endpoints with the same color, we can recolor the endpoints of one non-common edge of that cycle instead. So we must optimize the number of odd cycles that share their pair of vertices having the same color with another odd cycle.

In the fourth subtask, each new edge breaks the "same" cycle into two new cycles, so we can think about it as a sequence of cycles $C_0, \ldots, C_M$, where adjacent cycles share exactly one common edge. In this case, we can greedily pair the adjacent odd cycles in the sequence to have an optimal coloring.

In the general case, we shall notice that the new edges create a tree structure where the cycles are the nodes and the newly drawn edges are the adjacency relations between the nodes. See, e.g., the following example, where the image on the left will be converted to the tree on the right.





If we stick to the idea that each even cycle won't have adjacent vertices with the same color, then removing the nodes of even cycles breaks this tree into a forest where each tree consists of only nodes of odd cycles. In the above example, node B will be removed and the forest consists of three trees, each with a single node.

The number of adjacent vertices with the same color in each tree can be optimized by solving a maximum matching in a tree problem (e.g., using dynamic programming; see also this problem: CSES - Tree Matching). This solves the full problem.

We must also consider how to implement this efficiently. We can implement each step in the above solution, noting that the tree structure can be constructed in linear time by starting from a vertex and iterating over the original $0, 1, \ldots, N-1$ cycle simulating DFS: when we reach an endpoint of a newly drawn line, we either push it to a stack and start a new cycle, or close the current cycle depending whether we encountered the same line before. This gives an $O(N)$ solution as the maximum matching is also linear.

We note that $O(N \log N)$ implementations where the coloring is maintained using range-update data structures like BIT to perform flipping of the alternating coloring of cycles might be easier to implement and shall also pass.

Esoteric Bovines (`esotericbovines`)

Author: Bucă Mihnea-Vicențiu

Developer: Bucă Mihnea Vicențiu

Solution

Given two lists of integers, a and b, both of length N, the task is to find the K-th smallest value from the set of bitwise XOR combinations between elements of the two lists, where each combination is formed by XOR-ing an element from list a with an element from list b.

The order in which we will process the two lists of numbers does not matter.

Solving the second subtask involves brute-forcing the problem by computing the XOR value for each pair of elements from lists a and b, resulting in N^2 combinations. Then, either sorting these values or utilizing an algorithm to find the K-th smallest element can identify the K-th value. Complexity $\mathcal{O}(N^2 log(n))$ or $\mathcal{O}(N^2)$

For tackling the fourth subtask, a binary search approach to the answer is adopted. The question is how do we count the number of pairs with a bitwise XOR less than a given value x? Well to start, one of the lists, say b, is inserted into a trie data structure. This approach mirrors the algorithm used to find the minimal or maximal XOR (as seen in the third subtask). So for each element a_i in a, the algorithm counts how many elements b_j in b exist such that the XOR of a_i and b_j is less than x. The complexity of the algorithm is $\mathcal{O}(N \log(N) \log(\mathrm{MAX}))$, where MAX is the maximum value in the lists. It's worth noting that this process can also be optimized to run in $\mathcal{O}(\mathrm{MAX} \log(N))$. Since this optimization is not necessary to resolve the task, I'll leave it as an exercise for the reader to explore this optimization further.

For the intended solution of the problem, we can observe from the previous subtask that binary search isn't necessary. Instead, we can recover bits of the answer one by one by descending the trie level by level and maintaining pairs of trie nodes along with corresponding elements from list a that will provide us with the correct higher bits of the answer.

At each step, we find the number of paths whose next bit equals 0. To do this, we iterate over the list of pairs. For every pair, there are two ways to obtain 0 as the next bit of the result: the corresponding bits of the values in the trie and elements in a need to be both 0s or both 1s. We compute the number of paths as a product of the number of values in the corresponding subtrees and the number of elements in a that have these properties. If the sum is greater than or equal to K, then the next bit will be equal to 0; otherwise, it will be equal to 1, in this case, we need to reduce K by this number before proceeding to the next level.

The number of pairs at most quadruples when we go one level deeper, but in total we will encounter the same node at most 4 times in the pairs (twice with the first element in the pair and twice with the second element in the pair). Complexity $\mathcal{O}(N \log(MAX))$ or in our case $\mathcal{O}(60 \cdot N)$

Geometric Mean (`geometricmean`)

Author: Péter Gyimesi

Developer: Péter Csorba

Solution

We will need the prime factorization of our N numbers. The prime factorization of V can be done in $O(\sqrt{V})$ time. As now $V \approx N^2$ this is $O(N)$ for one number. So for N numbers, it can be done in $O(N^2)$ time. Using the Sieve of Eratosthenes is a little bit slower, because of the precalculation costs, but should fit in the time limit as well. For each number, we only need the exponents modulo 4, and if this is 0 we do not need to keep that prime factor. For example, $2^8 3^6$ is just 3^2 now.

For an integer x let $\mathrm{inv}(x)$ denote the smallest integer such that $x \cdot \mathrm{inv}(x)$ is a 4th power. For example $\mathrm{inv}(2^7 3^5) = \mathrm{inv}(2^3 3^1) = 2^1 3^3$. If we have the prime factorization of x then this can be quickly computed.

We want to find the $(i < j < k < \ell)$ quadruples. We will go through the array V with k and store the possible values of $\mathrm{inv}(V[i] \cdot V[j])$ for all $i < j (< k)$ pairs in a *map*. (We use the precalculated prime factorization here.) After that, we will check the possible values of $(k <)\ell$, and for each $k < \ell$ pair we can find $V[k] \cdot V[\ell]$ and from the *map* find the number of complementing pairs.

The final complexity will be $O(N^2 \log N)$ using a search tree (*map*), or $O(N^2)$ using a *hashed* data structure (unordered_map). Despite the large time limit: hopefully an $O(N^3)$ solution can *not* pass.

Remark. This problem is closely related to SET (the card game)! There a card can be represented by a vector in $\mathbb{Z}_3^4$, as for each card there are 3 options for the *color*, *shape*, *shading* and the *number* of the symbols. The 3 options can be represented by $\mathbb{Z}_3$ $(0, 1, 2)$ and a card is a 4-dimensional vector: (color, shape, shading, number). Three cards; three vectors u, v, w form a set iff $u + v + w$ is the null vector $(0, 0, 0, 0) \in \mathbb{Z}_3^4$ where $+$ is the coordinate-wise sum modulo 3. In this abstract setting, it is clear that for any two SET cards u, v there is a unique card w such that u, v, w form a set, namely $w = -(u + v)$, where $-$ is coordinate-wise the negative modulo 3.

In our setting, we want to count SETs in $\mathbb{Z}_4^\infty$. Any positive integer n can be written as $n = 2^{\alpha_2} 3^{\alpha_3} 5^{\alpha_5} 7^{\alpha_7} \ldots$ where we list all primes in increasing order in this decomposition, and the exponent is 0 except for finitely many primes. Now n corresponds to the vector $(\alpha_2\%4, \alpha_3\%4, \alpha_5\%4, \alpha_7\%4, \ldots)$. For example $2024 = 2^3 3^0 5^0 7^0 11^1 13^0 17^0 19^0 23^1 \ldots$ corresponds to $(3, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, \ldots)$. Note that here different numbers correspond to the same vector (e.g. 2 and $32 = 2^5$). 4 numbers form a SET iff the sum of the corresponding vectors is the zero vector $(0, 0, 0, \ldots)$. Now 4 numbers have integer geometric mean iff they form a SET.

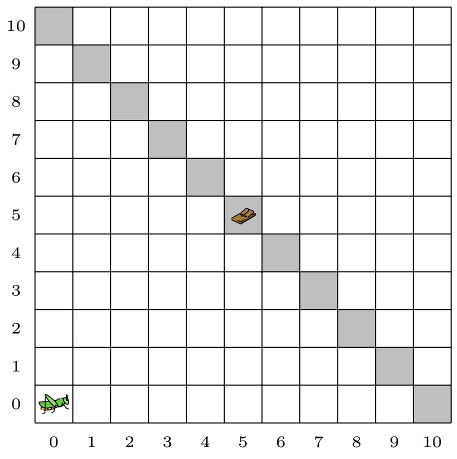
Carlo's Garden (grasshopper)

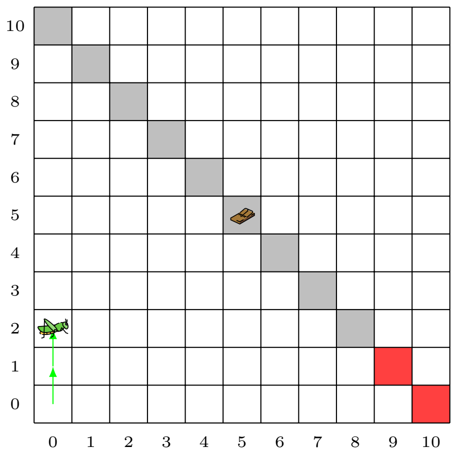
Author: Péter Csorba

Developer: Davide Bartoli

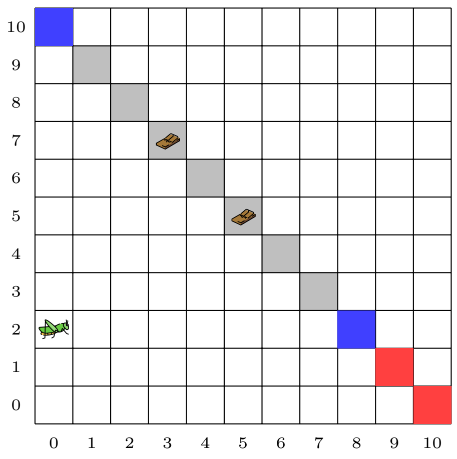
Solution

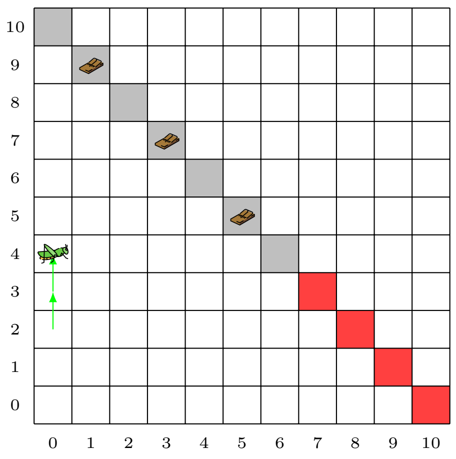
Let's look at the case $N = 2$ before the general case. Assume that we want to prevent the grasshoppers from reaching the $x + y = 10$ line. (There are 11 places there, but the grasshopper will get there before we can place 11 traps.) We put the first trap to $(5, 5)$. Now assume that the grasshopper jumps to $(0, 2)$ (via $(0, 1)$). Now we do not need to place traps on the red squares, so we only need to place 9 traps to catch, and hopefully, after each round, we need to place fewer.



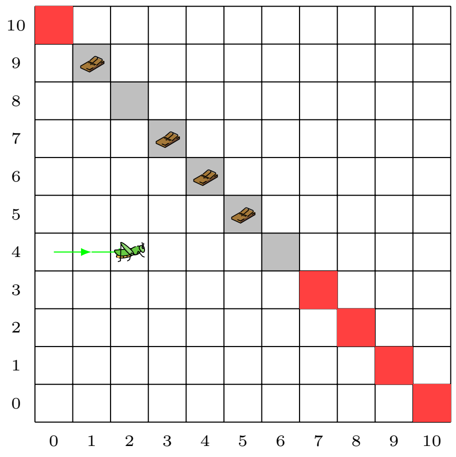


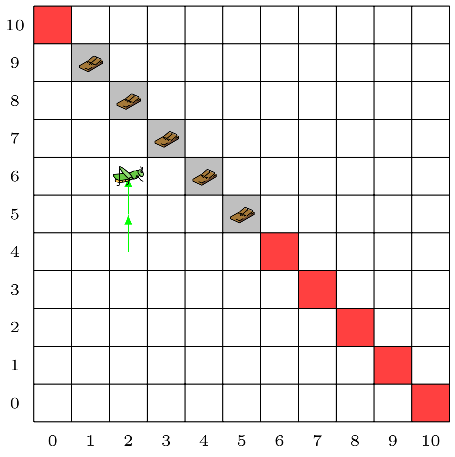
Let's place traps on every second square on the $x + y = 10$ line such that we place as far as possible from the boundary (blue) of the remaining 9 squares. So we place the second trap at $(3, 7)$. Assume now that the grasshopper jumps to $(0, 4)$, giving us 2 more red squares not to care about, and we place our trap at $(1, 9)$.





Assume that the grasshopper moves to $(2, 4)$, since we already placed traps on every second square we will get 1 more red square on average. We place our next trap on the closest place to the center of the remaining squares on the $x + y = 10$ line, to $(4, 6)$, and after that, we will place traps on every second square closest to the grasshopper. Assume that the grasshopper jumps to $(2,6)$, and we place a trap at $(2,8)$ blocking all places reachable for the grasshopper on the $x+y=10$ line, winning the game.





Let's look at the general case where N can be anything. After T seconds the grasshopper makes $T \cdot N$ jumps: arriving at the line $x+y=TN$[1], which contains $TN+1$ squares. After each second the grasshopper can not reach N of these points. In other words, the midpoint of the reachable subinterval can shift by at most $\frac{N}{2}$. So the idea is to first place a trap on the midpoint of this line, and after that place traps on the Nth, $2N$th, $3N$th neighbor such that the midpoint of the traps is as close to the midpoint of the grasshopper reachable subinterval as possible. Note that the midpoint of these traps will shift by $\frac{N}{2}$ in each round.

$\lceil \frac{T}{2} \rceil + 2$ seconds is used to place traps on every Nth squares of the reachable subinterval for the grasshopper.
We use the next $\lceil \frac{T}{4} \rceil + 2$ seconds to place a trap on every Nth squares (starting from one already not trapped point close to the middle) of the remaining reachable subinterval of the grasshopper.
...The last $\lceil \frac{T}{2^N} \rceil + 2$ seconds is used to place a trap on every remaining Nth square of the (already not trapped) remaining reachable subinterval of the grasshopper.
After these N stages we only have to wait (place traps in arbitrary places) and the grasshopper is CAUGHT.
This will work if

$$\left\lceil \frac{T}{2} \right\rceil + 2 + \left\lceil \frac{T}{4} \right\rceil + 2 + \ldots + \left\lceil \frac{T}{2^N} \right\rceil + 2 \leq T$$

Since $\frac{T}{2} + \frac{T}{4} + \ldots + \frac{T}{2^N} = T - \frac{T}{2^N}$ and $\lceil a \rceil \leq a + 1$ the above holds if

$$T - \frac{T}{2^N} + 3N \leq T$$

So we need $3N \leq \frac{T}{2^N}$, $3N \cdot 2^N \leq T$. If $N = 5$ then we get that $15 \cdot 2^5 = 480$ seconds is more than enough.

[1]Actually it is better to consider the $x+y=TN-N+1$ line...

Replace the Characters (`replacechar`)

Author: Bernard Ibrahimcha

Developer: Bernard Ibrahimcha

Solution

First, it is not optimal to replace the same character more than once. So now the problem becomes to select some characters that we want to replace, and the others will stay the same.

Now let's look at the characters that will stay the same. They must be non-decreasing when looking at them alone (imagine removing the characters that we apply the operation on).

Also, minimizing the number of operations is equivalent to maximizing the number of characters that will stay the same.

The problem now becomes to find a subsequence of the string of maximum length that is non-decreasing.

This problem can be solved using dynamic programming.

Let $dp_{i,j}$ be the length of the longest non-decreasing subsequence of the first i ($1 \le i \le n$) characters of the string while the last selected character in that subsequence being j ($0 \le j \le 25$).

The transitions are:

— we can either take s_i and go from $dp_{i,j}$ to dp_{i+1,s_i} and we add 1 to the length of the non-decreasing subsequence. We can do this transition only if $s_i \ge j$.

— or not take s_i and go from dpi, j to $dp_{i+1,j}$.

Then we can use a DFS to find the indices of the subsequence.

All the indices in the subsequence will stay the same and the others will be replaced with the last unreplaced character before them (or 'a' if there are no characters that were replaced before this one).

Triangle Counting (triangles)

Author: Péter Gyimesi

Developer: Stefan Dascalescu

Solution

For the small subtasks, you can implement brute-force-like solutions which rely on storing the edges in an adjacency matrix and processing the updates one by one.

A significant improvement that gets us up to 53 points consists of using bitsets to speed up the process of counting the triangles by finding for all pairs (i, j), how many values x exist such that there is an edge between x and i and an edge between j and x, which can result in updates being processed using bitwise AND.

To find the full solution, we can rely on the following observation: for any triplet of distinct vertices, if they do not form a triangle, then the outdegree of one of the vertices is 2, and the other two vertex cannot have this property. This makes it possible to count the number of triplets that are NOT triangles by looking at the outdegrees of every vertex.

There are $\frac{N\times(N-1)\times(N-2)}{6}$ triplets of distinct vertices. For each of the N vertices, the number of triplets in which a given vertex i is the one having outdegree 2 is $\frac{X_i\times(X_i-1)}{2}$, where X_i is the outdegree of i (i.e., the number of ways to select two distinct outgoing edges among all outgoing edges from i). Using this observation, the number of triangles initially is

$$\frac{N\times(N-1)\times(N-2)}{6}-\sum_{i=1}^{N}\frac{X_i\times(X_i-1)}{2},$$

and we can maintain the value of this expression over the updates by noticing that each update modifies the outdegree of two vertices, affecting only two terms in the summation. So the updates are done in constant time, and the total complexity is $O(N+Q)$.

TV Series (tvseries)

Author: Alessandro Bortolin

Developer: Tommaso Dossi

Solution

Denote by $dp_{i,d}$ the maximum number of days on which we participated in the conversation about the first i tv-series that we **finished** watching until at most day d. The answer we seek is the value of $dp_{N,D}$. Initially we have $dp_{0,d} = 0$ for all d. Let's iterate over the tv series and suppose that we already processed the first $i-1$ series and see how to take the i-th one into account.

The days on which this TV series is talked about are between $S[i]$ and $E[i]$. For each day j such that $S[i] \leq j \leq E[i]$, we know that if we finish watching the series before day j, then we will be able to talk about it for (at least) $E[i]-j+1$ days. For this to happen, we must start watching it before day $j-X[i]$. So if we start watching series i on day k, then $dp_{i,d}$ for each $d = k+X[i], k+X[i]+1, \ldots$ is at least $dp_{i-1,k} + E[i] - j + 1$.

If we iterate over each j from $S[i]$ to $E[i]$ and each k from day 1 to $j - X[i] - 1$ and do the update for every such d, then it may be too slow (with complexity $O(N \cdot D^2)$). Instead, let's just first set

$$dp_{i,k+X[i]} = \max_{S[i]\leq j\leq E[i], j>k+X[i]} dp_{i-1,k} + E[i] - j + 1 = dp_{i-1,k} + \max(0, E[i] - \max(S[i]-1, k+X[i]))$$

and then, for each $d = 2, \ldots, D$, let $dp_{i,d} = \max(dp_{i-1,d}, dp_{i,d-1}, dp_{i,d})$. This way the overall time complexity is $O(N \cdot D)$, which is sufficient.

Washington Distance (`washington`)

Author: Gabriella Blénessy

Developer: Stefan Dascalescu

Solution

To solve this problem, we need to be very careful about converting the coordinates given according to the Washington Distance by carefully treating each of the four cases depending on the quadrant we are at. The key thing is to be careful about the order of the letters as well as checking each direction and seeing if we have to flip the y coordinate or not.

Once we are done with the conversion, all we have to do is simply compute the Manhattan Distance by using the well-known formula.

Watch Towers (`watchtowers`)

Author: Áron Noszály

Developer: Apraham Jamil Avenian

Solution

For every watchtower, we must calculate two values: the minimal increase to see all watchtowers to its left and its right. The answer will be the maximum of these two values. Now let's just focus on the former.

So we would like to see the ith watchtower from the kth for all $i < k$. Now, for all j such that $i < j < k$ k must be "higher" than the $i - j$ line, otherwise j obstructs i from k. More formally we need

$$H_i + \frac{H_j - H_i}{j - i}(k - i) \leq H_k$$

This gives us a $O(N^3)$ solution, just check all (i, j, k) triplets.

Now, we can notice that it's sufficient to only look at adjacent watchtowers. Let's look at the line given by watchtowers $i - j$. $i - (i + 1)$ and $(j - 1) - j$ must both have a smaller slope than $i - j$, otherwise one of them is at least as "good" as $i - j$. So $i - (i + 1)$ goes "under" and $(j - 1) - j$ "over" this line. This means there must be a moment when we cross $i - j$ as we go from left to right. At that moment the watchtowers there cross $i - j$ and have a larger slope, i.e. they're adjacent and as good as $i - j$. So we can always find an adjacent pair of watchtowers that are as "good" as any non-adjacent pair of watchtowers. Using this observation the previous solution can be optimized to $O(N^2)$.

To optimize further, notice that at index k we want to calculate

$$\max_{i<k}(H_{i+1} - H_i)(k - i) = \max_{i<k}(H_{i+1} - H_i)k + c_i = \max_{i<k} m_i k + c_i$$

That is a maximum of linear functions. So we need a data structure that supports insertion of linear functions and querying the maximal value at a given position. This problem is usually called "dynamic convex hull trick" and can be solved in a lot of ways in $O(\log N)$ per operation. For example, using Li-Chao tree or a Binary Search tree. So to summarize, we have a solution with $O(N \log N)$ complexity.