Day 4 Contest

ByteDance Moscow Workshops Camp 2020

May 4, 2020

A. Airlines

Keywords: Eulerian cycle, greedy.

We are given an edge-colored graph, and have to find an Eulerian cycle so that no two consecutive edges in the cycle have the same color.

Let $d(v)$ be the degree of a vertex v. For the answer to exist, each vertex v must have even $d(v)$, and cannot have $> d(v)/2$ incident edges of the same color.

If this is true for all vertices, let's take each vertex v and divide its incident edges into pairs so that edges in each pair have different colors.

We can do this greedily. Among edges incident to v take two largest color groups c_1 and c_2, pair any edge of color c_1 with any edge of color c_2, and discard those edges. Initially no color group outnumbers all other groups combined, thus this will result in successfully pairing all edges incident to v.

For each pair of edges e_1 and e_2 put them next to each other in a cycle.

This breaks all edges of the graph into several disjoint cycles without consecutive edges of the same color. The rest is to merge them into a single valid cycle.

Suppose that two disjoint valid cycles C_1 and C_2 pass through a common vertex v. Without loss of generality we may assume that both cycles start and end at v.

For $i = 1, 2$ let e_i, e'_i be the first and the last edge of C_i. Edges e_i and e'_i must have different colors since C_i was a valid cycle.

Consider two ways to merge C_1 and C_2 into a single cycle:

- cycle C: go through C_1 followed by C_2. This is valid unless $c(e'_1) = c(e_2)$ or $c(e_1) = c(e'_2)$.
- cycle C': go through C_1 followed by C_2 in reverse. This is valid unless $c(e_1) = c(e_2)$ or $c(e'_1) = c(e'_2)$.

If both C and C' are invalid, then either $c(e_1) = c(e'_1)$ or $c(e_2) = c(e'_2)$, which is impossible. Thus, we can always merge two valid cycles with a common vertex into a single valid cycle.

Repeat merging while possible. In the end all resulting cycles are vertex-disjoint. But the graph is connected, thus we must finish with a single valid Eulerian cycle.

This solution can be implemented in linear time, so the **total complexity** is $O(n + m)$, but constraints are lax enough to allow less efficient implementations.

B. Battle Robots

Keywords: convex hull, scanline, optimization.

For two vectors v, u let $v \times u$ be the cross-product of v and u.

For points v, u, w define predicate $ccw(v, u, w)$ to be true when w lies to the left of the directed line vu (and the line is well-defined).

The predicate condition is equivalent to $(u - v) \times (w - v) > 0$.

Assume that in a set of points $p_1, \ldots, p_n$ no three points are collinear.

Two distinct points p_i, p_j are adjacent in the convex hull in counter-clockwise order if $ccw(p_i, p_j, p_k)$ is true for any $k \neq i, j$.

Let $p_i(t)$ be the location of the point i at time t. Let us assume that no three points always lie on the same line (in particular, no two points always share the same position).
Let's say that a time moment $t > 0$ is *special* when for some distinct i, j, k $(p_j(t) - p_i(t)) \times (p_k(t) - p_i(t)) = 0$. Note that $(p_j(t) - p_i(t)) \times (p_k(t) - p_i(t))$ cannot always be zero due to the assumption above.
Also include special moments 0 and ∞.
We can see that in a range $t \in (t_1, t_2)$ between two consecutive special moments the (cyclic) sequence $i_1, \ldots, i_k$ of vertices $p_{i_1}(t), \ldots, p_{i_k}(t)$ of the convex hull in counter-clockwise order does not change.

Since all coordinates are linear in t, for each i, j, k the value of $(p_j(t) - p_i(t)) \times (p_k(t) - p_i(t))$ is quadratic in t.
Solving $(p_j(t) - p_i(t)) \times (p_k(t) - p_i(t)) = 0$ produces at most two special moments.
It follows that there are $O(n^3)$ special moments in total.

If $p_0(t), \ldots, p_k(t) = p_0(t)$ is the counter-clockwise sequence of convex hull points in the range (t_1, t_2), then the area of the convex hull for any $t \in (t_1, t_2)$ is $A(t) = \frac{1}{2} \sum_{i=0}^{k-1} (p_i(t) \times p_{i+1}(t))$.
Similar to the above, $A(t)$ is quadratic when $t \in (t_1, t_2)$. Its minimum in (t_1, t_2) is either one of t_1 or t_2, or the global minimum of $A(t)$ (when it exists).
Optimizing $A(t)$ for each range (t_1, t_2), we can find the answer.

To account for degenerate configurations of points we can:
- get rid of duplicate points that always share the same position;
- modify $ccw(v, u, w)$ as follows: if v, u, w are collinear, $ccw(v, u, w)$ is true when $0 < (w - v) \cdot (u - v) < ||u - v||^2$ (that is, w is strictly inside the segment vu);
- consider special all moments when $ccw(p_i(t), p_j(t), p_k(t))$ changes value for some i, j, k, including the case when $p_i(t), p_j(t), p_k(t)$ are always on the same line.

Practically, we can construct convex hull from scratch for each range (t_1, t_2). This results in **total complexity** of $O(n^4 \log n)$, but can be optimized further.

C. Circles

Keywords: interactive, greedy.

Consider the set C_0 of all distinct circles with integer center coordinates and radius inside $[0, 20]^2$.
The answer is the sum of multiplicities of each circle in C_0 in the given configuration.
Suppose that a point p_0 is covered by exactly one circle c_0 in C_0. Since all circles are distinct, we can always find suitable p_0 and c_0.
Now, by asking p_0 we find out the multiplicity of c_0.
Let $C_1 = C_0 - c_0$. We can now adjust the answer to any other query p to find the total multiplicity of circles in C_1 covering p.
Similarly, we can look for a point p_1 covered by a unique circle $c_1 \in C_1$ to reduce the problem to $C_2 = C_1 - c_1$, and so on.
With this we can solve the problem in C_0 queries and $O(|C_0|^2)$ time.

To find p_i and c_i we can simply try all points in the square with a small enough step.

D. Davy Jones Tentacles

Keywords: DP, BFS.

Suppose that we've done i keystrokes before time $t + 1$. Consider time moments $0, \ldots, t$.
At each of these moments, consider three possible types of events:
- 0 — no keystroke;
- 1 — a keystroke was done by a tentacle x, but it's not the latest keystroke for x;
- 2 — a keystroke was done by a tentacle x, and it's the latest keystroke for x.

Note that this sequence of 0/1/2 infers for each stroke $1, \ldots, i$ when it was done, and which tentacle did it.

We now want to determine which tentacles can do the next keystroke $i+1$ at time $t+1$.
A tentacle x can do the next keystroke if it can reach (r_{i+1}, c_{i+1}) from its last keystroke, that is, from its respective occurence of an event 2 (or from its original position).
Numbering of tentacles does not matter, thus we can renumber them in order of their last keystrokes.
Also, for each event 1 we do not care which tentacle did the stroke, since it does not affect its ability to do stroke $i+1$.
Thus, we do not need to remember x for events of type 1 and 2 at all.
Further, we do not care about events happening before $t+1-\max(h,w)$. Indeed, any tentacle that did its last stroke before that time can now reach any key.
Thus, the situation is completely described by i, t, and the sequence S of $\leqslant \max(h,w)-1$ last event types (0/1/2).
Let us prepend S with 0's to make its length always be $\max(h,w)-1$.

Let $dp_{i,S}$ be the smallest t such that the situation (i,t,S) is reachable.
Possible transitions from this situation are:
• Press nothing at time t.
This appends a 0 to S (and erases its first entry).
• Do the keystroke $i+1$ with a tentacle x that made its last keystroke at time $t-d$ with $d \leqslant |S|$.
This is only possible if the tentacle can reach (r_{i+1}, c_{i+1}) in time. Note that i and S provide enough information to figure out which keystroke was the last for x.
S is appended with a 2, and the previous 2 for x is replaced with 1 (and the first entry of S is erased).
• Do the keystroke $i+1$ with a tentacle that made its last keystroke before $t-|S|$.
This is only possible if the keystroke $i+1$ can be made at all starting at original position, and not all k tentacles made their last keystrokes in the last $|S|$ moments, that is, S contains less than k 2's.
S is appended with a 2 (and the first entry of S is erased).

If's hard to find $dp_{i,S}$ with DP since there are cyclic dependencies between states.
However, we can construct a graph with vertices corresponding to all (i,S), and directed edges for transitions described above.
If $h \geqslant w$, the number of distinct S is 3^{h-1}, thus the graph has $O(n3^h)$ vertices and $O(n3^h \cdot h)$ edges.
Now all $dp_{i,S}$ are shortest path lengths from the initial position to (i,S), which can be found with BFS.
The answer is the smallest $dp_{n,S}$ over all S.
The **total complexity** is $O(n3^{\max(w,h)} \cdot \max(w,h))$.

E. Exploring the Space

Keywords: Gaussian elimination.

For a given collection of M vectors in $\mathbb{R}^n$ we need to find for each vector whether it can be represented as a linear combination of all others.
Let A be the $N \times M$ matrix, where *columns* contain coordinates of all the given vectors.
Apply Gaussian elimination to obtain its reduced row echelon form. In this form:
• non-zero rows precede zero rows;
• positions of leading non-zero elements of non-zero rows are distinct and increasing;
• a column with the leading non-zero element of a row has all other elements equal to 0.
Note that row operations of Gaussian elimination are changes of the common basis of the vectors, thus any operation preserves the answer.

Let us erase all zero rows from A. There no zero columns since all initial vectors are non-zero.
Suppose that a row i contains a single non-zero element $A_{i,j}$. Since $A_{i,j}$ is leading for the row i, the column j also does not have any non-zero elements.
It follows that column j is not a linear combination of other columns. We can also erase the row i and column j without affecting the answer.

In the remaining matrix, consider a random linear combination of all columns without leading elements.
There is a unique way to add this combination with columns with leading elements to obtain a zero vector.
The resulting combination (with probability 1) has non-zero coefficient for each column, since each of the remaining rows contains at least two non-zero elements, at least one of them non-leading.
This combination allows to represent any remaining column as a combination of all other remaining columns.

The **total complexity** of Gaussian elimination for an $N \times M$ matrix is $O(N^2M)$.

F. Funny Graphs

Keywords: bipartite matching.

If the desired degree of a vertex exceeds the largest degree in the graph, there clearly is no answer.
Let $V(d)$ be the set of all vertices with degree d. Initially for some d we have $V(d) \neq \varnothing$, and the set of all vertices $V = V(d-1) \cup V(d)$.
Let's try to remove some edges so that we have $V(d-1) \neq \varnothing$, $V = V(d-2) \cup V(d-1)$ in the remaining graph. If this is always possible, then we can obtain a solution in all remaining cases by decreasing d until done.

First, while there are any edges inside $V(d)$, we can greedily remove them to move pairs of vertices from $V(d)$ to $V(d-1)$. After this, $V(d)$ is an independent set (that is, there are no edges inside $V(d)$).
Now, if $V(d) = \varnothing$, then $V = V(d-1)$ and we are done. Otherwise, consider the induced bipartite subgraph H with halves $V(d)$ and $V(d-1)$.
Let us find a matching in H covering all vertices of $V(d)$. If we remove all edges of such matching, all remaining degrees will be $d-1$ and $d-2$, and $V(d-1) \neq \varnothing$, so we will be done.

The existence of such a matching follows from (a form of)
Hall's theorem. There is a matching covering the left half L of a bipartite graph G **iff** for all subsets $A \subseteq L$ we have $|N(A)| \geqslant |A|$, where $N(A)$ is the set of all distinct neighbours of vertices in A.
To see that the latter condition holds, observe that any vertex in $V(d)$ has degree d in H since there no edges inside $V(d)$.
On the other hand, any vertex in $V(d-1)$ has degree at most $d-1$ in H.
Thus for any $A \subseteq V(d)$, $d|A| = \sum_{v \in A} deg_H(v) \leqslant \sum_{u \in N(A)} deg_H(u) \leqslant (d-1)|N(A)|$, and $|N(A)| \geqslant |A|$.
Since the desired matching exists, we can successfully find it with Kuhn's algorithm and proceed with $V = V(d-1) \cup V(d-2)$.

There will be at most n steps of decreasing d.
Complexity-wise, the hardest part is to find a bipartite matching on each step of the process.
Technically, we then have **total complexity** $O(n^4)$, however, Kuhn's algorithm will most likely not achieve the worst-case complexity in this case.

G. Guards of Black and White

Keywords: Ford-Fulkerson theorem, network flows.

Let $M = nC + 1$, where C is the largest city population. Then, put the cost of putting a tower to a city v with population c_v to be $c'_v = M - c_v$.
The total cost of a valid solution (set of vertices with towers) P is then equal to $cost(P) = M|P| - \sum_{v \in P} c_v$.
Minimizing $cost(P)$ prioritizes minimizing $|P|$, and then maximizing $\sum c_v$. Thus, we are looking for the solution with smallest $cost(P)$.

The graph is bipartite. Let L and R be its parts.
In a solution P, let L_1 be the set of vertices with towers in L, and $L_2 = L \backslash L_1$. Define R_1, R_2 similarly.
In a valid solution P we cannot have edges between L_2 and R_2. The cost of this solution is then equal to $\sum_{v \in L_1 \cup R_1} c'_v$.

Construct a network as follows:
- create a source s and a sink t;
- for all $v \in L$ create an edge $s \to v$ of capacity c'_v;
- for all $u \in R$ create an edge $u \to t$ of capacity c'_u;
- for all edges vu with $v \in L$, $u \in R$ create an edge $v \to u$ of capacity $\infty > nM$.

Consider any cut (S, T) with $s \in S$, $t \in T$, $S \sqcup T = V$ (the disjoint union of S and T contains all vertices). Associate this cut with a solution $L_1 = L \cap T$, $L_2 = L \cap S$, $R_1 = R \cap S$, $R_2 = R \cap T$, $P = L_1 \cup R_1$.
The capacity of the cut is then equal to the sum of:
- c'_v for each vertex $v \in (L \cap T) = L_1$;
- c'_u for each vertex $u \in (R \cap S) = R_1$;
- ∞ for each edge vu of the original graph crossing the cut, that is, $v \in (L \cap S) = L_2$, $u \in (R \cap T) = R_2$.

Then, the capacity of the cut is $\geqslant \infty$ if the resulting solution P is invalid, otherwise the capacity is exactly $cost(P)$.
Thus, $\min cost(P)$ is equal to the minimum $s - t$ cut capacity.
By **Ford-Fulkerson theorem**, the minimum $s - t$ cut capacity is equal to maximum $s - t$ flow volume.
Further, the left part S of the cut can be found as the set of vertices reachable from s in the residual network. With this, we can restore the solution.

The complexity depends on the choice of a maximum flow algorithm. Dinic should be fine, but something like Ford-Fulkerson+scaling can also pass.

H. How to Pack String

Keywords: greedy methods, dynamic programming
Sort f-s in non-decreasing order. Then, we should assign codes $c_1, c_2, \ldots, c_n$ in such a way that:

- $1 \leqslant c_1 \leqslant c_2 \leqslant \ldots \leqslant c_n \leqslant L$ (if $f_i \geqslant f_{i+1}$ and $c_i > c_{i+1}$ then swapping of c_i and c_{i+1} decrease total length);
- for any i, $2^{c_i} - \sum_{j=1}^{i} 2^{c_i - j}$ should be at least 0;
- $\sum_{i=1}^{n} f_i * c_i$ should be minimal possible.

To do that, consider the following process. Suppose that we go along array and assign c_i for $i = 1, 2, \ldots, n$. At each moment of time, we have variable l, $1 \leqslant l \leqslant L$, and we can choose one of two options:

1. increase l by one, if $l \leqslant L$;
2. set c_i to l and increase i by one.

Initially, $i = l = 1$. To check whether we can set c_i to l or not, we will use variable $r := 2^l - \sum_{j=1}^{i-1} 2^{l - c_j}$; informally, r is a maximum number of strings of length c_i we can add to current set (each string of length c_j "blocks" exactly 2^{l-c_j} strings of length l). Initially, $r = 2$. If we use first option, then r is multiplied by 2; otherwise, we decrement r. Second option can be used only in case $r > 0$.

Note that if $r \geqslant n$ at some moment of time then using first option doesn't make sense anymore; then we can implement the solution as forward dynamics $dp[i][l][r]$, $1 \leqslant i \leqslant n + 1$, $1 \leqslant l \leqslant L$, $0 \leqslant r \leqslant 2n$. From each state, we have at most 2 transitions; so the total complexity of solution is $O(n^2 L)$.

I. Integer Pairs

Keywords: math, inclusion-exclusion principle
Factorize $X := p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$; then replace X by number $p_1 p_2 \ldots p_k$ without changing the answer. Note that $2 * 3 * 5 * 7 * 11 * 13 * 17 * 19 * 23 > 10^8$, so $k \leqslant 8$.

Let $f(y)$ be equal to number of pairs (i, j) such that $i < j$ and $a_i - a_j$ is a multiple of y. Then according to inclusion-exclusion principle, the answer can be found as $f(p_1) + f(p_2) + \ldots + f(p_k) - f(p_1p_2) - f(p_1p_3) - \ldots - f(p_{k-1}p_k) + f(p_1p_2p_3) + f(p_1p_2p_4) + \ldots + f(p_{k-2}p_{k-1}p_k) + \ldots + (-1)^{k-1}f(p_1p_2 \ldots p_k)$. But how to find $f(y)$?

Obviously, $a_j - a_i$ is divided by y iff $a_j \mod y = a_i \mod y$. Build hashmap M, and let $M[a]$ be number of elements a_i of the array such that $a_i \mod y = a$, $0 \leqslant y < a$ (of course we don't store zeroes). Then, $f(y)$ is $\sum_{\text{a is a key of M}} \binom{M[a]}{2}$ and can be found in $O(n)$. Total complexity of the solution is $O(\sqrt{X} + n * 2^k)$.

J. Journey for Treasure

Keywords: geometry, theory of probability

Workable algorithm to find a point p is a following cycle repeating enough number of times:

- Suppose that G is a convex polygon, and we now that $p \in G$;
- Let c be a center of mass of G; it's a known property that any line containing c divides G to two convex polygons of equal area;
- Make queries from point c. Answer for each query is a segment of length $2E$ of possible angles on a 360-degrees-circle; repeat queries until intersection of segments we have is more than 180 degrees;
- build a line l such that all possible direction are in one selfplane; replace G by intersection of G and this selfplane.

Then, just print mass center of current G.

K. King of Renovations

Keywords: min-cost-max-flow

Let $G = (V, E)$ be a given graph. We will find min cost flow (not min cost **max** flow!) in the following network $G' = (V', E')$:

- for each $v \in V$, add new vertices v' and v'' in V'; also add source s and sink t to V'. So, $|V'| = 2n + 2$;
- for each $v \in V$, connect s with v' by two edges: one with capacity 1 and cost $-\infty$ and one with capacity ∞ and cost 0;
- for each $v \in V$, connect v'' with t by two edges: one with capacity 1 and cost $-\infty$ and one with capacity ∞ and cost 0;
- for each $(u, v, w) \in E$ connect u' with v'' by an edge with capacity 1 and cost w.

Strict proof of the fact that min cost of any possible flow plus $\infty * 2n$ is equal to the answer is obvious and left to the reader.

In fact, we should not make more than $2n$ iterations in classical greedy algorithm (in decomposition of any flow of volume $2n + 1$ or higher, cost of at least one path will be nonnegative, so we can avoid it). Each iteration can be made by Dijkstra's algorithm with Johnson potentials, except first we should run Ford-Bellman algorithm or linear DP (at the beginning, G' is acyclic). So, the total complexity is $O(n^3)$.

L. Letters Arrangement

Keywords: constructive, BFS

Let n be the length of s. If $n \leqslant 10$ then solve the problem using BFS; so suppose that $n \geqslant 11$. Also, suppose without loss of generality that the number of Bs is at most the number of As; so number of Bs is not more than $\lfloor n/2 \rfloor - 1$. Also, let's use 1-based indexation of characters.

If the number of $B's$ is at most 1 then the problem is solvable in at most 6 moves; the solution of this case is left to the reader. Now we suppose that there are at least two Bs.

First of all, spend at most 2 steps to move __ to the end of s. After that, do the following steps in a cycle while there are at least three Bs to the left of __:

- let i be such an index that $s[i] = s[i+1] = \_$;
- let j, $1 \leqslant j \leqslant i-3$ be such an index that $S[j+1] = B$;
- move $\_\_$ to j and $j+1$, and then to $i-1$ and i.

Each successful iteration of cycle adds one B to the end of S; so if we made k iterations then we made at most $2k + 2 \leqslant 2 * (\lfloor n/2 \rfloor - 1 - b) + 2 \leqslant n - 2b$ moves, where b is the number of Bs before $\_\_$. Let $i, i+1$ be indices of $\_$. In our case $b = 2$, so we can do at least 4 more moves; let $j_1 < j_2 < i$ be indices of two Bs. Using the fact that $n \geqslant 11$, we note that number of As is not less than 5, and we can consider the following cases:

1. $j_1 + 1 = j_2$: we move $\_\_$ to $j_1, j_1 + 1$ and achieve the goal; in other words, we move i to j_1;
2. $1 \leqslant j_1 < j_1 + 1 < j_2 < i - 2$: move i to $j_2 + 1$, j_1, $i - 2$ and j_2;
3. $1 < j_1 < j_1 + 1 < j_2 = i - 2$: move i to $j_1 - 1$ and $i - 1$;
4. $1 = j_1 < j_1 + 1 < j_2 = i - 2$: i should be at least 8, so we move i to $i - 2$, 2, i and 1;
5. $1 < j_1 < j_1 + 2 < j_2 = i - 1$: move i to $i - 2$ and $j_1 - 1$;
6. $j_1 = i - 3, j_2 = i - 1$: move i to $i - 5$, $i - 2$, i, $i - 4$;
7. $j_1 = 1, j_2 = i - 1$: move i to $i - 2$, 1, $i - 1$.

The solution can be implemented in $O(n)$ if we store positions of Bs before $\_\_$ in an array or list (to find B between 1 and $i - 1$ we should check only, for example, last 3 Bs).

M. Maze Generation

Keywords: meet-in-the-middle.

A slower solution is to try every possible maze. For each maze allowing a path from the upper-left corner to the bottom-right one, it's easy to find its probability to be generated, and the required probability for each cell can be calculated afterwards. However, the number of possible mazes is 2^{NM}, that's a bit too much.

Let's use meet-in-the-middle to optimize the solution. Split the map with a vertical line in the middle. For both parts of the map, try all $2^{NM/2}$ possible configurations independently.

To know whether a path from the upper-left to the bottom-right corner exists, what we need to know for the configurations of both parts is:

- whether cells in the column neighboring to the dividing line contain walls;
- which of those cells that do not contain walls belong to the same connected components in their part of the map;
- which of these components is connected to the upper-left (bottom-right) cell.

This way, many configurations in each half of the map can be considered equivalent. The number of equivalence classes can be bounded with $2^N \cdot B_{\lceil N/2 \rceil} \cdot \lceil \frac{N}{2} \rceil$, where $B_{\lceil N/2 \rceil}$ is the $\lceil \frac{N}{2} \rceil$-th Bell number.

Finally, loop over the configuration class of the left part, and the configuration class of the right part. We can determine whether this pair of configuration classes allows a path from the upper-left to the bottom-right corner, which allows us to find the probability that a random maze allows such a path, and eventually all the required cell probabilities as well.