Petrozavodsk Camp, Day 1: Solutions

Jagiellonian University in Kraków

29.01.2021

Problem F
A Very Different Word

Author: Krzysztof Maziarz

Statement

Given two words s, t of the same length $n \leq 25\,000$ and a letter K, find a word x which contains K and is lexicographically between s and t, provided that s is lexicographically smaller than t.

Statement

Given two words s, t of the same length $n \leq 25\,000$ and a letter K, find a word x which contains K and is lexicographically between s and t, provided that s is lexicographically smaller than t.

Solution

Denote by $next(s)$ the next word (in the lexicographic order) after s which has length n, and $next^2(s) = next(next(s))$, etc. Notice that if we consider all words from $next(s)$ till $next^{26}(s)$, last letters of these words will cover the English alphabet. That means we can check only the next 26 words and we are guaranteed to find one which contains the letter K – unless we reach the word t first, in which case we need to output `NO`. Complexity: $O(n + \Sigma)$ or $O(n\Sigma)$ depending on how we check for the letter K.

Problem I
GCD vs. XOR

Author: Krzysztof Maziarz

Given a sequence of integers $a_1, \ldots, a_n$ we must find the number of pairs (a_i, a_j) satisfying $gcd(a_i, a_j) = a_i \oplus a_j$. We know that $a_1, \ldots, a_n \in [1, M]$.

Given a sequence of integers $a_1, \ldots, a_n$ we must find the number of pairs (a_i, a_j) satisfying $gcd(a_i, a_j) = a_i \oplus a_j$. We know that $a_1, \ldots, a_n \in [1, M]$.

First, as M is relatively small, we can solve the problem assuming that all numbers between 1 and M may appear in input. For every $x \in [1, M]$ we denote by $C[x]$ the number of x values in the input sequence.

Slower solution

Let us guess $d \in [1, M]$ and compute how many pairs (x, y) may satisfy $gcd(x, y) = x \oplus y = d$. Obviously, x must be a multiple of d.

Slower solution

Let us guess $d \in [1, M]$ and compute how many pairs (x, y) may satisfy $gcd(x, y) = x \oplus y = d$. Obviously, x must be a multiple of d.

We also guess $x = k \cdot d$ for $k \in [1, \frac{M}{d}]$. Now y is uniquely determined, as it must be $y = x \oplus d$. Then we can calculate y and check if $x < y$ and $gcd(x, y) = d$. If so, we have found $C[x] \cdot C[y]$ solutions.

Slower solution

Let us guess $d \in [1, M]$ and compute how many pairs (x, y) may satisfy $gcd(x, y) = x \oplus y = d$. Obviously, x must be a multiple of d.

We also guess $x = k \cdot d$ for $k \in [1, \frac{M}{d}]$. Now y is uniquely determined, as it must be $y = x \oplus d$. Then we can calculate y and check if $x < y$ and $gcd(x, y) = d$. If so, we have found $C[x] \cdot C[y]$ solutions.

This algorithm works in $\sum_{d=1}^{M} \frac{M}{d} \cdot \log M = \mathcal{O}(M \log^2 M)$.

Faster solution

We can do better: in fact, for given x and d, the only possible y is $x + d$.

Faster solution

We can do better: in fact, for given x and d, the only possible y is $x + d$.

Proof: let $2^j \leq d < 2^{j+1}$, so the j-th bit is the highest in d. Then the number $x \oplus d$ can only differ from x on j lowest bits, so $x \oplus d \leq x + 2^{j+1} - 1$, which means $x \oplus d < x + 2d$.

Faster solution

We can do better: in fact, for given x and d, the only possible y is $x + d$.

Proof: let $2^j \leq d < 2^{j+1}$, so the j-th bit is the highest in d. Then the number $x \oplus d$ can only differ from x on j lowest bits, so $x \oplus d \leq x + 2^{j+1} - 1$, which means $x \oplus d < x + 2d$.

It is then enough to check $y = x + d$. As $gcd(x, y) = d$ is now guaranteed, we only have to check if $x \oplus y = d$. This is $\mathcal{O}(M \log M)$.

Problem C
Jellyfish

Author: Marcin Briański

Statement

We are given a jellyfish[a] J with n vertices ($3 \leq n \leq 100\,000$). We say that $A \subseteq V(J)$ is *awesome* if for every $B \subseteq A$ there exists a connected subgraph of J which contains every vertex from B and does not contain any other vertex from A. What is the maximum possible size of an awesome A?

[a] A connected graph with equal number of vertices and edges.

Statement

We are given a jellyfish[a] J with n vertices ($3 \leq n \leq 100\,000$). We say that $A \subseteq V(J)$ is *awesome* if for every $B \subseteq A$ there exists a connected subgraph of J which contains every vertex from B and does not contain any other vertex from A. What is the maximum possible size of an awesome A?

[a]A connected graph with equal number of vertices and edges.

Solution

Let's try to derive a solution for a tree T, because trees are somewhat similar to jellyfish, but have easier structure.

Lemma

Let T be a tree with n vertices. The maximum size of an awesome subset of T is the number of leaves of T.

Lemma

Let T be a tree with n vertices. The maximum size of an awesome subset of T is the number of leaves of T.

Proof.

- If $n \leq 2$ then the statement is trivial. Otherwise, let's root our tree in a non-leaf vertex.
- On one hand, it's obvious that if we take any subset of leaves we can connect them without touching any other leaf. We can just throw out the other leaves from the tree and that's it.
- On the other hand, we can see that we can modify every optimal solutions such that all the leaves are occupied (easy exercise).
- If all the leaves are occupied, we cannot occupy any other vertex of the tree.

□

From this moment, it's easy to see that the answer for a jellyfish would be the number of leaves plus a small constant which depends on J.

Theorem

Let J be a jellyfish with n vertices. Let A be the number of leaves of J, and let B be the maximum number of consecutive vertices of the jellyfish cycle without proper subtree. Then the answer for the jellyfish is $max(3, A + min(B, 2))$.

From this moment, it's easy to see that the answer for a jellyfish would be the number of leaves plus a small constant which depends on J.

Theorem

Let J be a jellyfish with n vertices. Let A be the number of leaves of J, and let B be the maximum number of consecutive vertices of the jellyfish cycle without proper subtree. Then the answer for the jellyfish is $max(3, A + min(B, 2))$.

To prove this statement we need the following lemma.

Lemma

Let J be a jellyfish. There exists a maximum awesome subset S of J which contains all the leaves from J.

The proof is similar to the proof for trees and consists of several cases. For the sake of clarity, we omit the proof, but it's not too hard.

Theorem

Let J be a jellyfish with n vertices. Let A be the number of leaves of T, and let B be the maximum number of consecutive vertices of the jellyfish cycle without proper subtree. Then the answer for the jellyfish is $max(3, A + min(B, 2))$.

Proof.

- Obviously, we always can occupy 3 vertices on the jellyfish cycle. Unfortunately, we cannot occupy any other vertex in this case.
- From the previous lemma, we can occupy all the leaves of the jellyfish.
- If there is no vertex with empty subtree on a cycle then we can do nothing better.
- If there is a vertex with empty subtree then we can occupy such a vertex.
- If there are two consecutive vertices with empty subtrees we can occupy both of them.

□

Problem G
Cactus

Author: Krzysztof Maziarz

Statement

Given a cactus graph[a] with n vertices and a number k, count the number of colorings of the cactus vertices into k colors such that no two connected vertices have the same color.

[a]A connected graph where each vertex lies on at most one cycle.

Let’s solve progressively harder graph types.

Let's solve progressively harder graph types.

Path

Number of colorings is $path[n] = k \cdot (k-1)^{n-1}$

Let's solve progressively harder graph types.

Path

Number of colorings is $path[n] = k \cdot (k-1)^{n-1}$

Cycle

We take all colorings for a path, and then subtract colorings where the first and last vertex has the same color: $cycle[n] = path[n] - cycle[n-1]$

Let's solve progressively harder graph types.

Path

Number of colorings is $path[n] = k \cdot (k-1)^{n-1}$

Cycle

We take all colorings for a path, and then subtract colorings where the first and last vertex has the same color: $cycle[n] = path[n] - cycle[n-1]$

What about a cactus?

First, lets remove all tree edges from the cactus, to get a collection of cycles and isolated vertices.

First, lets remove all tree edges from the cactus, to get a collection of cycles and isolated vertices.

For each such component we know the number of colorings, and we multiply all these numbers together.

First, lets remove all tree edges from the cactus, to get a collection of cycles and isolated vertices.

For each such component we know the number of colorings, and we multiply all these numbers together.

Each tree edge connects one components (either a cycle or a single vertex) to its parent component. Once we have colored the parent component, this will block a single color in a single vertex of the child component.

For each tree edge, we have to multiply our number of colorings by $\frac{k-1}{k}$ to account for the one blocked color.

Problem K

We apologize
for any inconvenience

Author: Krzysztof Kleiner

Statement

There are n stops and k tram lines, each serving some subset of the stops. Then one of the lines gets suspended, then another, and then another... After each suspension event, determine the largest[a] number of line changes necessary to travel from any stop to any other.

[a] Excluding the "infinity" values.

We create a graph with a vertex for each stop and a vertex for each line. We add an edge whenever a given line serves a given stop.

We create a graph with a vertex for each stop and a vertex for each line. We add an edge whenever a given line serves a given stop.

Observation

Travelling between two stops requires c changes $\iff$ their respective vertices are at distance $2(c+1)$.

We create a graph with a vertex for each stop and a vertex for each line. We add an edge whenever a given line serves a given stop.

Observation

Travelling between two stops requires c changes $\iff$ their respective vertices are at distance $2(c+1)$.

We can compute all those distances in $O((n+k)^3)$. We would just need to find an algorithm allowing us to accommodate the disappearing lines...

Which is precisely what the standard solution does anyway.

In its *i*-th iteration, the Floyd-Warshall algorithm computes the lengths of the shortest paths between **all** pairs of vertices, using only the first *i* vertices as **intermediate** points.

Which is precisely what the standard solution does anyway.

In its i-th iteration, the Floyd-Warshall algorithm computes the lengths of the shortest paths between **all** pairs of vertices, using only the first i vertices as **intermediate** points.

Solution

- Sort the vertices in the following order:
  - The vertices for all the stops,
  - The vertices for those lines which are never suspended,
  - The vertices for the remaining lines, in the reversed order of disappearing.
- Run Floyd-Warshall.

Complexity: $O((n+k)^3)$.

Solution

- Sort the vertices in the following order:
  - The vertices for all the stops,
  - The vertices for those lines which are never suspended,
  - The vertices for the remaining lines, in the reversed order of disappearing.
- Run Floyd-Warshall.

Notes:

- We need to take the maximum over the distances between *stop* vertices only, not the graph's diameter. It is possible for a *line*-*line* pair of vertices to be at a distance strictly larger than the maximum over the *stop*-*stop* pairs.

Solution

- Sort the vertices in the following order:
  - The vertices for all the stops,
  - The vertices for those lines which are never suspended,
  - The vertices for the remaining lines, in the reversed order of disappearing.
- Run Floyd-Warshall.

Notes:

- We need to take the maximum over the distances between *stop* vertices only, not the graph's diameter. It is possible for a *line*-*line* pair of vertices to be at a distance strictly larger than the maximum over the *stop*-*stop* pairs.
- The first n iterations actually compute paths of length 2 only. By performing this step separately, we can reduce $O((n+k)^3)$ to $O((n+k)^2 \cdot k)$, but this wasn't needed to fit within the time limit.

Problem M
Social Justice

Author: Krzysztof Kleiner

Statement

Social justice is when nobody earns more than K times the average pay of the citizens. Given *n* people, we want to banish **as few people as possible** such that the remaining citizens form a socially just set (if there is more than one way to achieve this, we might choose any correct solution). Find those people who definitely cannot be allowed to stay in the town.

Statement

Social justice is when nobody earns more than K times the average pay of the citizens. Given n people, we want to banish **as few people as possible** such that the remaining citizens form a socially just set (if there is more than one way to achieve this, we might choose any correct solution). Find those people who definitely cannot be allowed to stay in the town.

The answer does not need to contain only the highest-earning or the lowest-earning citizens:

$$1, 1, 1, 10, 300, 300, 300$$

Sort the citizens by salaries.

Observation

If there exists a socially just subset of size m, then:

- there exists a socially just subset of size m which is an interval, Sort the citizens by salaries.

Observation

If there exists a socially just subset of size m, then:

- there exists a socially just subset of size m which is an interval,
- there exist such subsets for every $m' < m$.

Proof: Removing the lowest-earning citizen is always safe: the average pay increases (or stays the same), while the maximal pay does not change. Similarly, if our subset is not an interval (contains any "gaps"), then it is safe to remove the lowest-earning citizen and add someone from the gap instead.

$$max_i\ A_i \leq K \cdot \sum_i A_i/m$$

Therefore, we are able to find the largest possible size of social justice in $O(n \log n)$. We can use binary search and the fixed-length window moving, or the variable-length window moving.

Therefore, we are able to find the largest possible size of social justice in $O(n \log n)$. We can use binary search and the fixed-length window moving, or the variable-length window moving.

Observation

Let us fix a citizen c. If there exists a socially just subset of size m containing c, then there exists such a subset of the form $J \cup c$, where J is an interval of elements larger than c.

Proof: Analogously as in the previous step (replace the lowest-earning citizen with someone from the "gap", but do not touch c).

Observation

Let us fix an interval J. We will try to find all citizens with salaries **smaller or equal** than the salaries in J, who could possibly be added to this interval. Then the inequality

$$max_i\ A_i \leq K \cdot \frac{\sum_i A_i}{m}$$

simplifies into a linear inequality:

$$max_J \leq K \cdot \frac{\sum J + A_c}{m}$$

$$m \cdot max_J - K \cdot \sum J \leq K \cdot A_c$$

where A_c is the variable, and all other terms are constants (as long as a particular J is fixed).

Observation

Let us fix an interval J. We will try to find all citizens with salaries **smaller or equal** than the salaries in J, who could possibly be added to this interval. Then the inequality

$$max_i\ A_i \leq K \cdot \frac{\sum_i A_i}{m}$$

simplifies into a linear inequality:

$$max_J \leq K \cdot \frac{\sum J + A_c}{m}$$

$$m \cdot max_J - K \cdot \sum J \leq K \cdot A_c$$

where A_c is the variable, and all other terms are constants (as long as a particular J is fixed).

So, for a given interval J of length $m-1$, we can simply binary search the last index of the citizen that can be added to J and the set remains just.

Wrap-up:

- We find m: the largest size of a socially just subset.
- On the array of sorted salaries, we move a window (J) of length $m - 1$.
- At each position, we binary search the smallest element c which could be added to J to obtain a socially just set. If any such element is found (i,e. if it's not inside J nor is larger than the elements of J), we mark J, c and everything in between as an interval of citizens who have a chance to stay in the town.
- Then we compute the sum of those intervals. Any citizen not covered by the intervals is someone who must be banished.

Complexity: $O(nlogn)$.

Problem E
Archer Vlad

Author: Daniel Goc

Statement

We are given constant C and a sequence (x_i, y_i) of n pairs of positive integers. Find an angle α such that a projectile shot from point $(0, 0)$ at this angle with initial velocity C will fly above all the points (x_i, y_i). Output a possible value of $tan(\alpha)$.

Statement

We are given constant C and a sequence (x_i, y_i) of n pairs of positive integers. Find an angle α such that a projectile shot from point $(0, 0)$ at this angle with initial velocity C will fly above all the points (x_i, y_i). Output a possible value of $tan(\alpha)$.

Solution

Firstly, let us consider the case $n = 1$. We start with a lemma:

Statement

We are given constant C and a sequence (x_i, y_i) of n pairs of positive integers. Find an angle α such that a projectile shot from point $(0,0)$ at this angle with initial velocity C will fly above all the points (x_i, y_i). Output a possible value of $tan(\alpha)$.

Solution

Firstly, let us consider the case $n = 1$. We start with a lemma:

Lemma

For $n = 1$ the set of possible values for $tan(\alpha)$ is the interval

$$\left(\frac{C^2 - \sqrt{\Delta}}{g \cdot x_1}, \frac{C^2 + \sqrt{\Delta}}{g \cdot x_1}\right)$$

where $\Delta = C^4 - g^2 x_1^2 - 2gC^2 y_1$.

Proof

- Suppose we shot a projectile at an angle α giving it an initial horizontal velocity of $C \cdot cos(\alpha)$ and initial vertical velocity of $C \cdot sin(\alpha)$. When the x-coordinate of projectile reaches x_1 then its y-coordinate (denoted by y_p) becomes:

$$y_p = x_1 \cdot tan(\alpha) - \frac{g}{2}\left(\frac{x_1}{C \cdot cos(\alpha)}\right)^2$$

- Writing $cos(\alpha)$ in terms of $tan(\alpha)$ we transform the inequality $y_p > y_1$ into:

$$gx_1^2 tan^2(\alpha) - 2C^2 x_1 tan(\alpha) + gx_1^2 + 2C^2 y_1 < 0$$

- The rest is solving quadratic inequality.

Solution

So for $n = 1$ we can find all the possible values of $tan(\alpha)$ in $O(1)$ time.

Solution

So for $n = 1$ we can find all the possible values of $tan(\alpha)$ in $O(1)$ time.

But now notice that for $n > 1$ we can divide the problem into many $n = 1$ problems, and then take the intersection of all the solutions.

Solution

So for $n = 1$ we can find all the possible values of $tan(\alpha)$ in $O(1)$ time.

But now notice that for $n > 1$ we can divide the problem into many $n = 1$ problems, and then take the intersection of all the solutions.

This gives us the following algorithm:

Algorithm

- For each point (x_i, y_i) calculate the interval of possible values for $tan(\alpha)$.
- Find the intersection of all these intervals, which will again be an interval.
- Output any element from the resulting interval.

Problem B
(Almost) Fair Cake-Cutting

Author: Wiktor Kuropatwa

Statement

Given a square-shaped cake and n lines (cuts), each dividing the square into two non-zero-area parts. For each cut, Alice can choose one of the sides (a half-plane) and eat all the cake at this side. Determine the largest area of the cake which can be covered with an optimal choice of the half-planes.

There are 2^n possible choices of the half-planes.

However, we can see that the lines divide the plane (and the square) into (at most) $T(n+1) = T(n) + n + 1$ pieces.
$T(n) = O(n^2)$.

There are 2^n possible choices of the half-planes.

However, we can see that the lines divide the plane (and the square) into (at most) $T(n+1) = T(n) + n + 1$ pieces.
$T(n) = O(n^2)$.

So, for a large enough n (it's easy to check that this is satisfied for $n \geq 3$ actually), there must always exist at least one choice of half-planes such that their intersection is empty. Alice can choose her solution by taking the opposite half-plane in each of the equations - this way, the aforementioned empty intersection is precisely what will be left for Bob to eat. Alice is therefore able to cover the whole plane and the answer is always 100%.

There are 2^n possible choices of the half-planes.

However, we can see that the lines divide the plane (and the square) into (at most) $T(n+1) = T(n) + n + 1$ pieces.
$T(n) = O(n^2)$.

So, for a large enough n (it's easy to check that this is satisfied for $n \geq 3$ actually), there must always exist at least one choice of half-planes such that their intersection is empty. Alice can choose her solution by taking the opposite half-plane in each of the equations - this way, the aforementioned empty intersection is precisely what will be left for Bob to eat. Alice is therefore able to cover the whole plane and the answer is always 100%.

What remains to be done is to handle the cases of $n = 2$ and $n = 1$.

For $n = 2$:

- If the lines intersect outside of the square, the answer is also 100%.

For $n = 2$:

- If the lines intersect outside of the square, the answer is also 100%. Including if they are parallel

For $n = 2$:

- If the lines intersect outside of the square, the answer is also 100%. Including if they are parallel (maybe equal).

For $n = 2$:

- If the lines intersect outside of the square, the answer is also 100%. Including if they are parallel (maybe equal).
- Otherwise, the lines divide the square into 4 pieces, of which we need to find the one with the smallest area and give it to Bob.

For $n = 2$:

- If the lines intersect outside of the square, the answer is also 100%. Including if they are parallel (maybe equal).
- Otherwise, the lines divide the square into 4 pieces, of which we need to find the one with the smallest area and give it to Bob.

For $n = 1$, we need to find the smaller of the two pieces and give it to Bob, which with a little bit of care we can do using the same code we already needed to implement for $n = 2$.

Problem D
Flat Organization

Author: Krzysztof Kleiner

Statement

Given a tournament (a full directed graph) with positive weights, we want to reverse some of the edges to obtain a strongly connected graph (i.e. one where any vertex is reachable from any other). Find a solution which minimizes the sum of weights of reversed edges.

Statement

Given a tournament (a full directed graph) with positive weights, we want to reverse some of the edges to obtain a strongly connected graph (i.e. one where any vertex is reachable from any other). Find a solution which minimizes the sum of weights of reversed edges.

Solution 1

- Compute the graph of strongly connected components (SCCs).
- Create bidirectional edges between SCCs, with weight 0 in one direction and a positive weight in the opposite direction.
- Run Dijkstra from the bottom SCC to the top SCC.
- Reverse all the edges **with positive weights** on the shortest path found by Dijkstra.

Complexity: $O(n^2 \log n)$

Statement

Given a tournament (a full directed graph) with positive weights, we want to reverse some of the edges to obtain a strongly connected graph (i.e. one where any vertex is reachable from any other). Find a solution which minimizes the sum of weights of reversed edges.

Solution 1

- Compute the graph of strongly connected components (SCCs).
- Create bidirectional edges between SCCs, with weight 0 in one direction and a positive weight in the opposite direction.
- Run Dijkstra from the bottom SCC to the top SCC.
- Reverse all the edges **with positive weights** on the shortest path found by Dijkstra.

Complexity: $O(n^2 \log n)$
And don't forget about $n = 2$.

Statement

Given a tournament (a full directed graph) with positive weights, we want to reverse some of the edges to obtain a strongly connected graph (i.e. one where any vertex is reachable from any other). Find a solution which minimizes the sum of weights of reversed edges.

Solution 2

- Do not compute the graph of SCCs, just find any vertex in the bottom SCC. For example, the **first vertex in the DFS postorder** or the **vertex with the highest in-degree**.

Statement

Given a tournament (a full directed graph) with positive weights, we want to reverse some of the edges to obtain a strongly connected graph (i.e. one where any vertex is reachable from any other). Find a solution which minimizes the sum of weights of reversed edges.

Solution 2

- Do not compute the graph of SCCs, just find any vertex in the bottom SCC. For example, the **first vertex in the DFS postorder** or the **vertex with the highest in-degree**.
  (This is true **only** because the graph is a tournament!)

Statement

Given a tournament (a full directed graph) with positive weights, we want to reverse some of the edges to obtain a strongly connected graph (i.e. one where any vertex is reachable from any other). Find a solution which minimizes the sum of weights of reversed edges.

Solution 2

- Do not compute the graph of SCCs, just find any vertex in the bottom SCC. For example, the **first vertex in the DFS postorder** or the **vertex with the highest in-degree**.
  (This is true **only** because the graph is a tournament!)
- Analogously, find any vertex in the top SCC.
- Then follow solution 1 on the original graph rather than on the SCC graph.

Solution 3

- Compute the graph of SCCs and observe that it must be a path.
- Use dynamic programming to find the shortest route from the bottom to the top of the path.

Solution 3

- Compute the graph of SCCs and observe that it must be a path.
- Use dynamic programming to find the shortest route from the bottom to the top of the path.
- When performing the relaxation step for some edge (v, u) in the SCC graph, we need to use the formula:

$$dist[u] = weight[v, u] + minimumDist(u + 1, \dots, v),$$

rather than

$$dist[u] = weight[v, u] + dist[v],$$

because the shortest route might go through some edges that do not require reversing. We can compute it using a simple segment tree ($O(n^2 logn)$), or by choosing an order of relaxations which permits recomputing the prefix minimums on the fly ($O(n^2)$).

Problem J
Civilizations

Author: Krzysztof Maziarz

Problem

We're given an n x n field divided into n^2 unit squares; each square has an owner civilization and a value. For civilization p, we define its length of borders l_p and sum-of-values w_p. There are q events where a unit square changes owners, after each event determine the maximum value of $A \cdot w_p + B \cdot l_p + C \cdot w_p \cdot l_p$ (A, B, C are different for each query).

We can think of civilizations as points (l_p, w_p). Every change of owner for a single unit square impacts the values for the old owner, new owner, and owners of adjacent squares. In total, $\mathcal{O}(1)$ points change.

We can think of civilizations as points (l_p, w_p). Every change of owner for a single unit square impacts the values for the old owner, new owner, and owners of adjacent squares. In total, $\mathcal{O}(1)$ points change.

If not for the $C \cdot w_p \cdot l_p$ part, we could think of the convex hull of our set of points. But handling the full scoring function for arbitrary points and updates seems tricky...

We can think of civilizations as points (l_p, w_p). Every change of owner for a single unit square impacts the values for the old owner, new owner, and owners of adjacent squares. In total, $\mathcal{O}(1)$ points change.

If not for the $C \cdot w_p \cdot l_p$ part, we could think of the convex hull of our set of points. But handling the full scoring function for arbitrary points and updates seems tricky...

Is there anything special about the set of points (l_p, w_p)?

The range of values for l_p and w_p are large, that's not going to help. But...

The range of values for l_p and w_p are large, that's not going to help. But...

Observation

The sum of values of l_p is $\mathcal{O}(n^2)$.

The range of values for l_p and w_p are large, that's not going to help. But...

Observation

The sum of values of l_p is $\mathcal{O}(n^2)$.

Fact

If $x_1 + \cdots + x_k = m$, then there are only $\mathcal{O}(\sqrt{m})$ *distinct* values x_i.

So, we get a crucial observation: while there may be $\mathcal{O}(n^2)$ civilizations, there will only be $\mathcal{O}(n)$ different values of l_p at any given time.

We can rewrite our scoring as

$$A \cdot w_p + B \cdot l_p + C \cdot w_p \cdot l_p = B \cdot l_p + (A + C \cdot l_p) \cdot w_p$$

If l_p is fixed, it's optimal to choose either the largest or smallest w_p (depending on the sign of $A + C \cdot l_p$).

We can rewrite our scoring as

$$A \cdot w_p + B \cdot l_p + C \cdot w_p \cdot l_p = B \cdot l_p + (A + C \cdot l_p) \cdot w_p$$

If l_p is fixed, it's optimal to choose either the largest or smallest w_p (depending on the sign of $A + C \cdot l_p$).

Lets store all civilizations grouped by l_p; within a single group sorted by w_p.

We can rewrite our scoring as

$$A \cdot w_p + B \cdot l_p + C \cdot w_p \cdot l_p = B \cdot l_p + (A + C \cdot l_p) \cdot w_p$$

If l_p is fixed, it's optimal to choose either the largest or smallest w_p (depending on the sign of $A + C \cdot l_p$).

Lets store all civilizations grouped by l_p; within a single group sorted by w_p.

To answer a query, we iterate through all groups of civilizations and check either the minimum or maximum element.

We can rewrite our scoring as

$$A \cdot w_p + B \cdot l_p + C \cdot w_p \cdot l_p = B \cdot l_p + (A + C \cdot l_p) \cdot w_p$$

If l_p is fixed, it's optimal to choose either the largest or smallest w_p (depending on the sign of $A + C \cdot l_p$).

Lets store all civilizations grouped by l_p; within a single group sorted by w_p.

To answer a query, we iterate through all groups of civilizations and check either the minimum or maximum element.

Complexity: $\mathcal{O}(n^2 + q \cdot n)$.

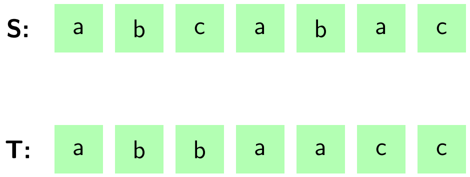
Problem A
Edit Distance Yet Again

Author: Adam Polak

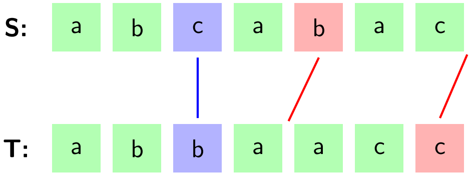
Statement

Given two strings $S = s_1 \dots s_n$, $T = t_1 \dots t_m$ and number k ($n, m \leq 1\,000\,000, k \leq 1000$), compute the edit distance between the strings or report that it's larger than k.

We can present any solution as a matching between S and T – we match s_i and t_j if s_i is going to be changed into t_j. We match t_j with the position s_i/s_{i+1} if t_j needs to be inserted between s_i/s_{i+1}, and similarly for deletion.



We can present any solution as a matching between S and T – we match s_i and t_j if s_i is going to be changed into t_j. We match t_j with the position s_i/s_{i+1} if t_j needs to be inserted between s_i/s_{i+1}, and similarly for deletion.



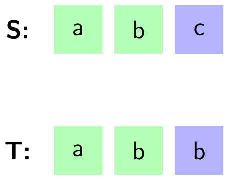
A matching of size r corresponds to the edit distance r between S and T. If we determine the best matching, we can easily retrieve the required sequence of operations.

Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.

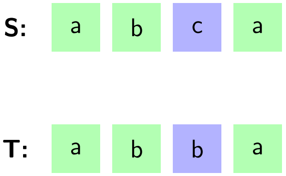
Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.



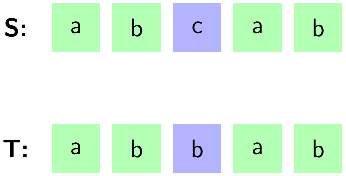
Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.



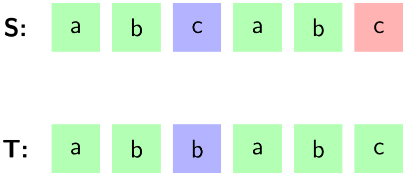
Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.



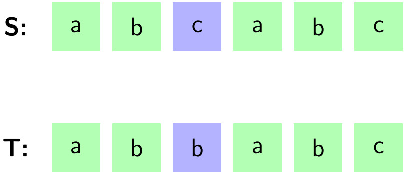
Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.



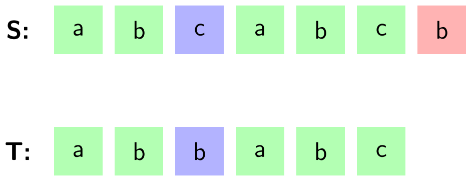
Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.



Observation 1

After each operation, we can pair the following identical symbols of S and T greedily, for as long as possible.



Let us rephrase our question: imagine that we perform a sequence of alternating operations:

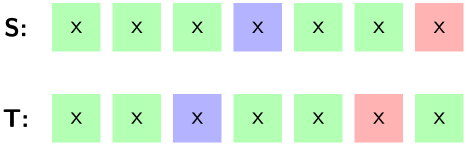
- Greedily pair letters of S and T one-to-one.
- Perform a single edit: either add a letter to T (INSERT), a letter to S (DELETE) or a letter to both sequences (REPLACE), at a cost 1.

We must reach (n, m) – the end of both sequences – with minimal cost (not exceeding k). So let us denote by A_r the set of pairs that we can reach with cost exactly r.

First, it is obvious that we never reach any pair (i, j) with $|i - j| > k$. But there's more:

Observation 2

If there are two pairs $(i, i + d), (i', i' + d) \in A_r$ and $i > i'$, then we may forget $(i', i' + d)$, as the optimal solution does not need it.



This is true because in every solution we can replace steps leading to $(i', i' + d)$ with "better" steps leading to $(i, i + d)$, and then proceed as we did before.

Solution idea: compute all the "important" pairs in A_r. We perform a move $A_r \to A_{r+1}$ the following way:

1. For any pair $(i, j) \in A_r$, we add $(i+1, j)$, $(i, j+1)$ and $(i+1, j+1)$ to A_{r+1}.
2. For every pair $(i', j') \in A_{r+1}$ we prolong the sequences, for as long as we can – to the largest possible $(i'+q, j'+q)$ such that $S[i'..i'+q] = T[j'..j'+q]$.
3. Remove all unnecessary pairs from A_{r+1} – this is done by simple sorting.
   (At most $2k+1$ pairs will be left, as $i - j \in \{-k, -k+1, \ldots, k\}$.)

- For step 2 (greedy pairing), we need Longest Common Prefix of S and T. We can do that using suffix array or hashes. . .

- For step 2 (greedy pairing), we need Longest Common Prefix of S and T. We can do that using suffix array or hashes...
- ...aaand we were somewhat nasty – suffix array with KMR is too slow AND the hashes modulo 2^{64} get *Wrong Answer*. So you need Karkkainen-Sanders...

- For step 2 (greedy pairing), we need Longest Common Prefix of S and T. We can do that using suffix array or hashes. . .
- . . . aaand we were somewhat nasty – suffix array with KMR is too slow AND the hashes modulo 2^{64} get *Wrong Answer*. So you need Karkkainen-Sanders. . .
- . . . just kidding, you can use hashes modulo two primes. (Probably, you could squeeze KMR/suffix tree/DBF, too).

- For step 2 (greedy pairing), we need Longest Common Prefix of S and T. We can do that using suffix array or hashes. . .
- . . . aaand we were somewhat nasty – suffix array with KMR is too slow AND the hashes modulo 2^{64} get *Wrong Answer*. So you need Karkkainen-Sanders. . .
- . . . just kidding, you can use hashes modulo two primes. (Probably, you could squeeze KMR/suffix tree/DBF, too).
- Total complexity is $\mathcal{O}(n + k^2 \log n)$.

However, keeping A_r as the set of pairs also tends to be very slow – it is better to frame it as dynamic programming algorithm:

$best[r][d]$ – the largest i such that $(i, i+d)$ is in A_r, for $r = 1, 2, \dots k$ and $d = -k, -k+1, \dots, k-1, k$.

Problem H
Social Distancing

Author: Marcin Briański

Statement

We are given a tree with n nodes and two independent[a] subsets of its vertices S_1 and S_2. In one move, we can change a single element of the first set into another vertex connected by an edge. Transform S_1 into S_2 by a sequence of $\mathcal{O}(n^2)$ moves such that all intermediate sets are also independent.

[a]Subset is independent if doesn't contain two vertices connected by an edge

Let's transform both S_1 and S_2 to some canonical form. If these forms are different, then the answer is NO; otherwise the answer is YES and we can obtain the solution by concatenating the sequence that canonicalizes S_1 with the one that canonicalized S_2 in reverse.

Let's transform both S_1 and S_2 to some canonical form. If these forms are different, then the answer is NO; otherwise the answer is YES and we can obtain the solution by concatenating the sequence that canonicalizes S_1 with the one that canonicalized S_2 in reverse.

How to define the canonical form of S?

Let's transform both S_1 and S_2 to some canonical form. If these forms are different, then the answer is NO; otherwise the answer is YES and we can obtain the solution by concatenating the sequence that canonicalizes S_1 with the one that canonicalized S_2 in reverse.

How to define the canonical form of S?

Intuition: lets root the tree, and construct the canonical form of S by pushing all elements of S as low as possible.

Formally: number vertices by decreasing distance from the root, breaking ties arbitrarily. We will construct the lexicographically smallest S' which is reachable from S.

Formally: number vertices by decreasing distance from the root, breaking ties arbitrarily. We will construct the lexicographically smallest S' which is reachable from S.

We go through the vertices in our new order; when we see a vertex $v \notin S$, we try to modify S to include v, but making sure we don't break the previously "fixed" vertices which we consider frozen.

Formally: number vertices by decreasing distance from the root, breaking ties arbitrarily. We will construct the lexicographically smallest S' which is reachable from S.

We go through the vertices in our new order; when we see a vertex $v \notin S$, we try to modify S to include v, but making sure we don't break the previously "fixed" vertices which we consider frozen.

We run dfs from v to find a vertex in S that we could pull into v. Often we can pull the closest vertex $u \in S$, unless the next node on the $u - v$ path (lets call it x) has two children in S. In that case, u would like to move to x but the other child blocks it.

Therefore we need one more subroutine: pushing nodes away from v, to get rid of any blocking nodes (if possible). We do dfs from v, when we exit a node $u \in S$ we see if we could move u to one of its children; if so we choose arbitrarily and move it. This ensures that every $u \in S$ moves one step away from v (as long as its possible).

Therefore we need one more subroutine: pushing nodes away from v, to get rid of any blocking nodes (if possible). We do dfs from v, when we exit a node $u \in S$ we see if we could move u to one of its children; if so we choose arbitrarily and move it. This ensures that every $u \in S$ moves one step away from v (as long as its possible).

After such a push, if all $u \in S$ that we could pull into v are blocked by their sibling, its not possible to include v into S. Otherwise we pull in any u and continue.

Therefore we need one more subroutine: pushing nodes away from v, to get rid of any blocking nodes (if possible). We do dfs from v, when we exit a node $u \in S$ we see if we could move u to one of its children; if so we choose arbitrarily and move it. This ensures that every $u \in S$ moves one step away from v (as long as its possible).

After such a push, if all $u \in S$ that we could pull into v are blocked by their sibling, its not possible to include v into S. Otherwise we pull in any u and continue.

In both push-dfs and pull-dfs, we never go into the frozen (previously considered) vertices. This is okay, because the frozen vertices are "at the bottom" of the tree, i.e. we never have to go through a frozen vertex to reach a non-frozen vertex.

To sum up, we consider all n nodes in a loop, for each we do a push-dfs followed by a pull-dfs, both can perform $\mathcal{O}(n)$ moves.

To sum up, we consider all n nodes in a loop, for each we do a push-dfs followed by a pull-dfs, both can perform $\mathcal{O}(n)$ moves.

We get a solution with $\mathcal{O}(n^2)$ time complexity and the same number of moves.

To sum up, we consider all n nodes in a loop, for each we do a push-dfs followed by a pull-dfs, both can perform $\mathcal{O}(n)$ moves.

We get a solution with $\mathcal{O}(n^2)$ time complexity and the same number of moves.

With a clear picture of what should be done, this problem is actually surprisingly easy to implement, and the model solution fits under 200 lines.

Problem L
Patrol Drone

Author: Krzysztof Maziarz

Statement

We are given two decks D_1 and D_2 containing cards with letters {N, E, S, W}. A deck represents a path in Cartesian plane, and we are given a starting/ending point (C_x, C_y) for both decks. In each moment, we only have access to the 2 top elements of deck D_1, which we can swap, remove (if they show opposite directions) or we can add 2 opposite cards on top. Lastly, we can put the top card of D_1 to the bottom, simultaneously moving in the direction written on that card. Without ever crossing point $(0, 0)$, transform deck D_1 into deck D_2, or determine that it's impossible.

Solution

The first idea is to convert both paths in $O(n^2 + m^2)$ time and at most $n^2 + m^2$ (plus some linear, insignificant stuff) operations into something simpler. For each path the result of conversion will be a (possibly empty) path going around point $(0, 0)$, completely contained in $3x3$ square centered at $(0, 0)$, and starting at the point $(0, 1)$.

Solution

The first idea is to convert both paths in $O(n^2 + m^2)$ time and at most $n^2 + m^2$ (plus some linear, insignificant stuff) operations into something simpler. For each path the result of conversion will be a (possibly empty) path going around point $(0, 0)$, completely contained in 3x3 square centered at $(0, 0)$, and starting at the point $(0, 1)$.

Since we will conduct both conversions independently, let us narrow our focus down to just deck D_1.

Suppose that the path of D_1 is completely contained in some rectangle (we pick the smallest one) of size $(a+1) \times (b+1)$. We will now introduce a simple algorithm that decreases the size of this rectangle.

Suppose that the path of D_1 is completely contained in some rectangle (we pick the smallest one) of size $(a+1) \times (b+1)$. We will now introduce a simple algorithm that decreases the size of this rectangle.

Algorithm

Suppose that $a > 0$ and let the greatest y-coordinate of some station on that path be $y_{max} > 1$. To decrease a by 1 do the following:

- Start by going to some station with y-coordinate smaller than y_{max}.
- Now make a full cycle of n moves.
- Whenever encountering a card pointing to some square with y-coordinate equal to y_{max}, use swap. Then move.
- Whenever encountering two opposite cards on top, remove them.

It's a simple exercise to show that this algorithm reduces the amount of cards in deck by at least two. Also, repeating it k times works in $O(k \cdot n)$ time and utilizes at most $2nk + n$ moves, so approximately n "moves per card". Hence, in the first part of solution we try to reduce a and b to the minimum by rotating the Cartesian plane and applying the algorithm to the maximum.

In the next part we simply walk around the path and remove from the top any opposing cards we encounter, until there are none we can remove. It can be shown that this part takes at most $\frac{nk}{2} + n$ moves, where k is the amount of cards we have removed. This is about $\frac{n}{2}$ "moves per card".

In the next part we simply walk around the path and remove from the top any opposing cards we encounter, until there are none we can remove. It can be shown that this part takes at most $\frac{nk}{2} + n$ moves, where k is the amount of cards we have removed. This is about $\frac{n}{2}$ "moves per card".

After proper treatment of 2 cases that arise after executing the step above, the path will take exactly the form that we described at the beginning:

Structure

(possibly empty) path going around point $(0, 0)$, completely contained in $3x3$ square centered at $(0, 0)$, and starting at the point $(0, 1)$.

So suppose that we have converted decks D_1 and D_2 into their simpler forms E_1 and E_2 respectively. When is it possible to transform E_1 into E_2?

So suppose that we have converted decks D_1 and D_2 into their simpler forms E_1 and E_2 respectively. When is it possible to transform E_1 into E_2?

The answer is surprisingly simple:

Theorem

We can transform path E_1 into E_2 if and only if $E_1 = E_2$.

So suppose that we have converted decks D_1 and D_2 into their simpler forms E_1 and E_2 respectively. When is it possible to transform E_1 into E_2?

The answer is surprisingly simple:

Theorem

We can transform path E_1 into E_2 if and only if $E_1 = E_2$.

Intuitively speaking, it's because for any path P our operations cannot change how many times P circles the point $(0, 0)$. More formal proofs of this theorem lie in the field of topology.

We can see now see that the only problem left is to merge two sequences of moves, which we obtained when converting D_1 and D_2 into their simpler forms. That we can do by simply reversing (with proper treatment) the second sequence of moves and concatenating it with the first one. That concludes the algorithm.

We can see now see that the only problem left is to merge two sequences of moves, which we obtained when converting D_1 and D_2 into their simpler forms. That we can do by simply reversing (with proper treatment) the second sequence of moves and concatenating it with the first one. That concludes the algorithm.

Lastly, we have noted that on average we remove a card every n moves, so total amount of moves in the final sequence will be bounded above by $n^2 + m^2$ (with some linear stuff).

Problemsetters

Lech Duraj
Daniel Goc
Vladyslav Hlembotskyi
Krzysztof Kleiner
Krzysztof Maziarz
Władysław Raczek

Thank you for your attention!