Polish Collegiate Programming Contest 2021 Editorial

Jagiellonian University

7th November 2021

Problem H
Hidden password

Author: Krzysztof Maziarz

We are given string H_1 and we know that there exists string $H_2 : H_2 \neq H_1$ and integer $d : 0 < d < 26$ such that Caesar cipher with shift d transforms H_1 into H_2 and H_2 into H_1.

We are given string H_1 and we know that there exists string $H_2 : H_2 \neq H_1$ and integer $d : 0 < d < 26$ such that Caesar cipher with shift d transforms H_1 into H_2 and H_2 into H_1.

We see that after double encryption H_1 transforms into itself, and because $H_1 \neq H_2$ it follows that d must be equal to 13. This information is sufficient for obtaining H_2 as it is just H_1 encrypted with shift 13 Caesar ciper.

Problem D
Divided mechanism

Author: Daniel Goc

Task

We are given two rectilinear shapes on the plane and a sequence of instructions telling, that we should move one of them either up, down, left or right until it hits the other shape. We need to decide, if during the process we were able to move that part arbitrarily away from the other shape.

Both figures consist of number of cells small enough to bruteforce the simulation of the whole process.

Both figures consist of number of cells small enough to bruteforce the simulation of the whole process.

We move the part one by one unit, until we can. If this machine part got far enough from the other, we can state that the figures have been decoupled.

Both figures consist of number of cells small enough to bruteforce the simulation of the whole process.

We move the part one by one unit, until we can. If this machine part got far enough from the other, we can state that the figures have been decoupled.

To simplify the implementation, we can store the cells of the stationary part in the input array, and keep the cells of the moving part, for example, in a list.

Problem L
Lemurs

Author: Daniel Goc

Task

On a $n \times m$ map, there are some cells inhabited by lemurs. Each lair has a foraging area – every cell not further than k cells in taxicab metric from it. Given cells, in which lemurs are foraging, we have to decide, if there exists specification of lairs location corresponding to that foraging area.

Let's note, that if lemurs are **not** foraging somewhere, then any lair cannot be located closer or with distance equal to k from that cell. Let's cross out all such cells from the map, for example using BFS.

Can we place lemur lairs on the remaining cells, so that every marked cell is reachable from them? Anyhow we place them, the foraging area generated by them won't be "too large", it could only not span all the needed cells.

Therefore, we can wlog put lairs at **every** non-crossed out cells – such lair won't be ever inconsistent with our data. We place lairs everywhere, where it is possible and examine their total range – again, we can use BFS for that part, which gives linear solution to the whole problem.

Problem K
Kitten and Roomba

Author: Krzysztof Maziarz

Task

We are given a tree of size n and a sequence $a_1, a_2, ..., a_m$ of vertices that Roomba will visit (in that order). Initially, a kitten is sleeping in vertex c. When Roomba enters vertex with a cat, it will wake up and escape to a randomly chosen, neighbouring vertex. Calculate the expected number of times the kitten will be woken up by the robot.

Task

We are given a tree of size n and a sequence $a_1, a_2, \ldots, a_m$ of vertices that Roomba will visit (in that order). Initially, a kitten is sleeping in vertex c. When Roomba enters vertex with a cat, it will wake up and escape to a randomly chosen, neighbouring vertex. Calculate the expected number of times the kitten will be woken up by the robot.

Limits: $n \leq 1\,000\,000$, $m \leq 5\,000\,000$.

For every vertex v we will keep track of probability $p[v]$ that the cat is currently in this vertex. Initially we have $p[c] = 1.0$.

For every vertex v we will keep track of probability $p[v]$ that the cat is currently in this vertex. Initially we have $p[c] = 1.0$.

When Roomba enters vertex v, we add $p[v]$ to the result, we update probabilities for all d_v neighbours by adding $\frac{p[v]}{d_v}$, and finally, we set $p[v] = 0$.

If we implement it naively, then the operation of adding to v's neighbours will take $O(d_v)$ time, which is too much (the whole algorithm may take $\mathcal{O}(nm)$ time).

If we implement it naively, then the operation of adding to v's neighbours will take $O(d_v)$ time, which is too much (the whole algorithm may take $\mathcal{O}(nm)$ time).

To optimize it, for each vertex v we will keep two values: $p[v]$ and $lazy[v]$, where $lazy[v]$ is the value that we want to lazily add to v's children. To calculate the probability that the kitten is in the vertex u we will check value of $p[u] + lazy[parent[u]]$.

If we implement it naively, then the operation of adding to v's neighbours will take $O(d_v)$ time, which is too much (the whole algorithm may take $\mathcal{O}(nm)$ time).

To optimize it, for each vertex v we will keep two values: $p[v]$ and $lazy[v]$, where $lazy[v]$ is the value that we want to lazily add to v's children. To calculate the probability that the kitten is in the vertex u we will check value of $p[u] + lazy[parent[u]]$.

Now, adding to the neighbours consists of just two operations: adding $\frac{p[v]}{d_v}$ to $lazy[v]$ and $p[parent[v]]$.

If we implement it naively, then the operation of adding to v's neighbours will take $O(d_v)$ time, which is too much (the whole algorithm may take $\mathcal{O}(nm)$ time).

To optimize it, for each vertex v we will keep two values: $p[v]$ and $lazy[v]$, where $lazy[v]$ is the value that we want to lazily add to v's children. To calculate the probability that the kitten is in the vertex u we will check value of $p[u] + lazy[parent[u]]$.

Now, adding to the neighbours consists of just two operations: adding $\frac{p[v]}{d_v}$ to $lazy[v]$ and $p[parent[v]]$.

Complexity: $\mathcal{O}(n + m)$.

Problem J
Jungle Trail

Author: Krzysztof Maziarz

Task

We are given a $n \times m$ grid. Each square is either empty, blocked (impassable) or contains a den of snakes, either poisonous or benign (not poisonous).

We can *tap* a column/row. In that case all poisonous snakes in this column/row are turned to benign, and vice versa.

We have to tap some of the columns/rows, so that we can get from top-left corner, to the bottom-right one, moving only down or right, and visiting only empty squares or squares with benign snakes.

If there is no path from the top-left corner, to the bottom-right one that goes through non-blocked squares, then the answer is obviously “NO“. If there is such a path, we will show that the answer is always “YES“.

If there is no path from the top-left corner, to the bottom-right one that goes through non-blocked squares, then the answer is obviously "NO". If there is such a path, we will show that the answer is always "YES".

Key observation: in every step the path visits a new row or a new column.

If there is no path from the top-left corner, to the bottom-right one that goes through non-blocked squares, then the answer is obviously “NO“. If there is such a path, we will show that the answer is always “YES“.

Key observation: in every step the path visits a new row or a new column.

If we visit a square that contains poisonous snakes we can tap either the current column or row, so the current square turns into a den of benign snakes, and the squares that we have already visited remain untouched.

Problem C

Cake

Author: Krzysztof Maziarz

Task

We are given two arrays $2 \times n$ of integers. We want to transform the second one into the first one by preforming the operation: rotate a 2×2 square by 180 degrees. Compute the minimum number of operations required or decide that it is impossible.

Task

We are given two arrays $2 \times n$ of integers. We want to transform the second one into the first one by preforming the operation: rotate a 2×2 square by 180 degrees. Compute the minimum number of operations required or decide that it is impossible.

Limits: $n \leq 500\,000$.

Rotation by 180 degrees corresponds to swapping $t[0][i]$ with $t[1][i+1]$ and $t[1][i]$ with $t[0][i+1]$ for some i.

Rotation by 180 degrees corresponds to swapping $t[0][i]$ with $t[1][i+1]$ and $t[1][i]$ with $t[0][i+1]$ for some i.

Let's swap $t[0][i]$ and $t[1][i]$ for even i (in both arrays). After that change a rotation by 180 degrees corresponds to just swapping two consecutive columns.

Rotation by 180 degrees corresponds to swapping $t[0][i]$ with $t[1][i+1]$ and $t[1][i]$ with $t[0][i+1]$ for some i.

Let's swap $t[0][i]$ and $t[1][i]$ for even i (in both arrays). After that change a rotation by 180 degrees corresponds to just swapping two consecutive columns.

The problem simplifies to: given a sequence of pairs, transform it into another sequence by swapping consecutive elements.

Let's enumerate pairs in the target sequence with the numbers $1, 2, \ldots, n$ (we use all numbers even if there are duplicates in the sequence).

Let's enumerate pairs in the target sequence with the numbers $1, 2, ..., n$ (we use all numbers even if there are duplicates in the sequence).

If we enumerate pairs in the origin sequence with the numbers corresponding to them in the target sequence (be careful with duplicates), then the answer is *number of inversions* in a sequence (it is well known problem).

Let's enumerate pairs in the target sequence with the numbers $1, 2, \ldots, n$ (we use all numbers even if there are duplicates in the sequence).

If we enumerate pairs in the origin sequence with the numbers corresponding to them in the target sequence (be careful with duplicates), then the answer is *number of inversions* in a sequence (it is well known problem).

We can count the inversions using *merge sort*, segment tree or indexed set. Complexity $\mathcal{O}(n \log n)$.

Problem F
Fence

Author: Krzysztof Maziarz

Task

We are given a sequence $a_1, a_2, \ldots, a_n$. For a fixed integer b we *split* all the a_i's into $\lfloor \frac{a_i - 1}{b} \rfloor$ copies of a number b and $((a_i - 1) \mod b) + 1$.

Task

We are given a sequence $a_1, a_2, \ldots, a_n$. For a fixed integer b we *split* all the a_i's into $\lfloor \frac{a_i - 1}{b} \rfloor$ copies of a number b and $((a_i - 1) \mod b) + 1$.

Example: For $a = [4, 5, 6]$ and $b = 2$ we get $[2, 2, 2, 2, 1, 2, 2, 2]$.

Task

We are given a sequence $a_1, a_2, \ldots, a_n$. For a fixed integer b we *split* all the a_i's into $\lfloor \frac{a_i - 1}{b} \rfloor$ copies of a number b and $((a_i - 1) \mod b) + 1$.

Example: For $a = [4, 5, 6]$ and $b = 2$ we get $[2, 2, 2, 2, 1, 2, 2, 2]$.

For every b compute the sum of elements with odd indices in the resulting sequence.
Limits: $\sum a_i \leq 10^6$.

It turns out that to compute the answer we only need three values:

It turns out that to compute the answer we only need three values:

$[5, 4, 3, 2, 1]$

It turns out that to compute the answer we only need three values:

$[5, 4, 3, 2, 1]$
Number of elements in the sequence $\rightarrow$ 5.

It turns out that to compute the answer we only need three values:

$[5, 4, 3, 2, 1]$
Number of elements in the sequence $\rightarrow$ 5.
Sum of elements with odd indices $\rightarrow$ 9.

It turns out that to compute the answer we only need three values:

$[5, 4, 3, 2, 1]$
Number of elements in the sequence $\rightarrow$ 5.
Sum of elements with odd indices $\rightarrow$ 9.
Sum of elements with even indices $\rightarrow$ 6.

Given b we can compute the above values for a *split* of a_i in $O(1)$.

Given b we can compute the above values for a *split* of a_i in $O(1)$.
Let's build a segment tree over the sequence.

Given b we can compute the above values for a *split* of a_i in $O(1)$.
Let's build a segment tree over the sequence.
In every node we will keep the three values for the corresponding segment.
It is easy to compute the values for a node given the values in its children.

Given b we can compute the above values for a *split* of a_i in $O(1)$.
Let's build a segment tree over the sequence.
In every node we will keep the three values for the corresponding segment.
It is easy to compute the values for a node given the values in its children.

Iterating over $b = 1\ldots$ we need to update only leaves with corresponding $a_i \geq b$. The total number of updates equals $\sum_{i=1}^{n} a_i$ which yields the $O((\sum_{i=1}^{n} a_i) \cdot \log(n))$ complexity.

Given b we can compute the above values for a *split* of a_i in $O(1)$.
Let's build a segment tree over the sequence.
In every node we will keep the three values for the corresponding segment.
It is easy to compute the values for a node given the values in its children.

Iterating over $b = 1\ldots$ we need to update only leaves with corresponding $a_i \geq b$. The total number of updates equals $\sum_{i=1}^{n} a_i$ which yields the $O((\sum_{i=1}^{n} a_i) \cdot \log(n))$ complexity.

Bonus: solve it in $O(\sum_{i=1}^{n} a_i)$.

Problem I
Interesting numbers

Author: Team work

Statement

A sequence of integers $(a_1, a_2, \ldots a_n)$ and a number k are given. Let's consider a graph in which vertices i and j are connected iff $a_i \oplus a_j \leq k$. We are asked to find the size of a maximum clique in this graph.

Lets solve the problem using recurrence.

Lets solve the problem using recurrence.
How to solve the problem for a fixed k and a sequence of integers a_i, where $a_i \in [0, 2^g)$?

Case $g = 0$. The problem is trivial.

Case $g = 0$. The problem is trivial.

Case $k < 2^{g-1}$. If a_i and a_j differ on the $(g-1)$-th bit, then there is no edge between them, so we can independently solve the problem for the ranges: $[0, 2^{g-1})$ and $[2^{g-1}, 2^g)$.

Case $g = 0$. The problem is trivial.

Case $k < 2^{g-1}$. If a_i and a_j differ on the $(g-1)$-th bit, then there is no edge between them, so we can independently solve the problem for the ranges: $[0, 2^{g-1})$ and $[2^{g-1}, 2^g)$.

Case $k \geq 2^{g-1}$. If a_i and a_j don't differ on the $(g-1)$-th bit, then there exists an edge between them, so an edge may **not** exist only if $a_i < 2^{g-1}$ and $a_j \geq 2^{g-1}$ (i.e. a_i and a_j differ on the $(g-1)$-th bit).

We have two cliques such that some of the vertices of these cliques are connected. We can solve this problem in a few different ways:

We have two cliques such that some of the vertices of these cliques are connected. We can solve this problem in a few different ways:

We can solve this problem applying recurrence. Total complexity is $\mathcal{O}(n \log(max_i a_i))$.

We have two cliques such that some of the vertices of these cliques are connected. We can solve this problem in a few different ways:

We can solve this problem applying recurrence. Total complexity is $\mathcal{O}(n \log(max_i a_i))$.

Another approach is to notice that now we are looking for the maximum independent set in a bipartite graph.

We have two cliques such that some of the vertices of these cliques are connected. We can solve this problem in a few different ways:

We can solve this problem applying recurrence. Total complexity is $\mathcal{O}(n \log(max_i a_i))$.

Another approach is to notice that now we are looking for the maximum independent set in a bipartite graph.

Kőnig theorem: Size of a maximum independent set equals $|V|$ minus maximum matching in bipartite graphs.

In order to find the matching we can either modify matching algorithm to make it work efficiently for this specific bipartite graph...

In order to find the matching we can either modify matching algorithm to make it work efficiently for this specific bipartite graph...

or use a *segment tree* to compress the graph and then find the maximum flow in the resulting graph using any reasonable flow algorithm (i.e. Dinic).

Problem A
AMPPZ in the times of disease

Author: Krzysztof Maziarz

Task

Partition n points on the plane into k (non-empty) groups, such that the longest distance between points inside the same group is less than the shortest distance between points from different groups.

Task

Partition n points on the plane into k (non-empty) groups, such that the longest distance between points inside the same group is less than the shortest distance between points from different groups.

Note: you can assume, that the solution always exists!

Task

Partition n points on the plane into k (non-empty) groups, such that the longest distance between points inside the same group is less than the shortest distance between points from different groups.

Note: you can assume, that the solution always exists!

Limits: $n \leq 2\,000\,000$, $k \leq 20$.

Solution 1:

Take any point x_1, wlog add it to the group 1.

Solution 1:

Take any point x_1, wlog add it to the group 1.

Let x_2 be the farthest point from x_1 (in case of draws select any). It can be shown, that x_2 has to be in the different group than x_1, so let it be the group 2.

Solution 1:

Take any point x_1, wlog add it to the group 1.

Let x_2 be the farthest point from x_1 (in case of draws select any). It can be shown, that x_2 has to be in the different group than x_1, so let it be the group 2.

In general: if we have assigned points $x_1, ..., x_i$ into groups (respectively) $1, \ldots, i$ so far, then x_{i+1}, which maximizes minimal distance to any of $x_1, ..., x_i$ has to go into a new group.

Solution 1:

Take any point x_1, wlog add it to the group 1.

Let x_2 be the farthest point from x_1 (in case of draws select any). It can be shown, that x_2 has to be in the different group than x_1, so let it be the group 2.

In general: if we have assigned points $x_1, ..., x_i$ into groups (respectively) $1, \ldots, i$ so far, then x_{i+1}, which maximizes minimal distance to any of $x_1, ..., x_i$ has to go into a new group.

At the end, we assign each of the remaining $n - k$ points into a group represented by the nearest from $x_1, ..., x_k$.

Solution 1:

Take any point x_1, wlog add it to the group 1.

Let x_2 be the farthest point from x_1 (in case of draws select any). It can be shown, that x_2 has to be in the different group than x_1, so let it be the group 2.

In general: if we have assigned points $x_1, ..., x_i$ into groups (respectively) $1, \ldots, i$ so far, then x_{i+1}, which maximizes minimal distance to any of $x_1, ..., x_i$ has to go into a new group.

At the end, we assign each of the remaining $n - k$ points into a group represented by the nearest from $x_1, ..., x_k$.

Naive implementation achieves $\mathcal{O}(nk^2)$ complexity, but it can be easily sped up to $\mathcal{O}(nk)$.

Solution 2:

Take arbitrary $k+1$ points. Some two of them must be in the same group, so particularly, the closest two of them must be in the same group. Let's label them x and y.

Solution 2:

Take arbitrary $k+1$ points. Some two of them must be in the same group, so particularly, the closest two of them must be in the same group. Let's label them x and y.

So, we add edge (x, y), remove one of these points from the set, and we add a new point. We repeat this process for each remaining point. The connected components of resulting graph will generate required partition into groups.

Solution 2:

Take arbitrary $k+1$ points. Some two of them must be in the same group, so particularly, the closest two of them must be in the same group. Let's label them x and y.

So, we add edge (x, y), remove one of these points from the set, and we add a new point. We repeat this process for each remaining point. The connected components of resulting graph will generate required partition into groups.

The complexity of a naive implementation is $\mathcal{O}(nk^2)$, using std::sets we can improve the theoretical complexity to $\mathcal{O}(nk \log k)$ (in practice, it's really slow).

Problem G
Gebyte's Grind

Author: Krzysztof Maziarz

THEORETICAL COMPUTER SCIENCE
Jagiellonian University

Task

Consider a witcher with health points H and three kinds of objects:

- *beast*, which decreases hp by b_i (and kills if $H \leq b_i$);
- *inn*, which kills if $H < k_i$, and sets H to k_i otherwise; and
- *witch*, that sets $H = \max(H, c_i)$.

The witcher's trail is a sequence of n objects. We have to handle q operations:

- *changes*: one object changes into another
- *queries*: we start at the position l_i with hp H_0, and we wonder to which $r_i \geq l_i$ we are able to get, without dying

Limits: $n \leq 2\,000\,000$, $q \leq 4\,000\,000$.

We can think about every object as a function transforming starting health into end witcher's health (or 0 if he dies).

We can think about every object as a function transforming starting health into end witcher's health (or 0 if he dies).

We can assemble functions for single objects easily, a sequence of many objects is also a function, however, more complicated...

We can think about every object as a function transforming starting health into end witcher's health (or 0 if he dies).

We can assemble functions for single objects easily, a sequence of many objects is also a function, however, more complicated...

...but can it be very complicated?

Turns out that an *arbitrary* sequence of objects after composing gives a function having following representation:

$$f(x) = \begin{cases} 0 & \textit{for } x \in [0, a] \\ y & \textit{for } x \in [a+1, b] \\ x - b + y & \textit{for } x \in [b+1, +\infty] \end{cases}$$

Turns out that an *arbitrary* sequence of objects after composing gives a function having following representation:

$$f(x) = \begin{cases} 0 & \textit{for } x \in [0, a] \\ y & \textit{for } x \in [a+1, b] \\ x - b + y & \textit{for } x \in [b+1, +\infty] \end{cases}$$

Proof: every base object is a special case of such function and a composition of such functions returns a function in that form too.

We maintain a segment tree, in every node keeping composition of objects from the corresponding base segment as a function in the above form (three numbers a, b, c).

We maintain a segment tree, in every node keeping composition of objects from the corresponding base segment as a function in the above form (three numbers a, b, c).

The composition of two nodes can be implemented in constant time.

We maintain a segment tree, in every node keeping composition of objects from the corresponding base segment as a function in the above form (three numbers a, b, c).

The composition of two nodes can be implemented in constant time.

Change operation translates to leaf update and recalculation of $\mathcal{O}(\log n)$ nodes. To answer a query, we start in a leaf corresponding to l_i, firstly climbing up, and later traversing down (also $\mathcal{O}(\log n)$ time).

We maintain a segment tree, in every node keeping composition of objects from the corresponding base segment as a function in the above form (three numbers a, b, c).

The composition of two nodes can be implemented in constant time.

Change operation translates to leaf update and recalculation of $\mathcal{O}(\log n)$ nodes. To answer a query, we start in a leaf corresponding to l_i, firstly climbing up, and later traversing down (also $\mathcal{O}(\log n)$ time).

Final complexity: $\mathcal{O}((n+q)\log n)$.

Problem E
Epidemic

Author: Krzysztof Kleiner

Task

You are given n people, every person might be infected or not. Then k consecutive events of the following form occur:

1. Group of people have a meeting - if any of them were infected, then everybody from this group becomes infected (and remain infected until the end of their lives).
2. Some person is tested and receives a negative result.
3. Some person is tested, receives a positive results and is put under quarantine.
4. You receive a query: Is it possible to prove that people put under quarantine are the only ones infected?

You need to be able to answer all the queries *online*.

Limits: $n \leq 500\,000$, $k \leq 1\,000\,000$.

We can represent current state of our knowledge as a directed acyclic graph. Initially, it consists of n isolated vertices, on the i-th of then we put a pawn corresponding to the i-th person.

We can represent current state of our knowledge as a directed acyclic graph. Initially, it consists of n isolated vertices, on the i-th of then we put a pawn corresponding to the i-th person.

- We keep a set *possibly_infected* of all people who might be infected and are not under quarantine. Initially, this set contains all n people.

We can represent current state of our knowledge as a directed acyclic graph. Initially, it consists of n isolated vertices, on the i-th of then we put a pawn corresponding to the i-th person.

- We keep a set *possibly_infected* of all people who might be infected and are not under quarantine. Initially, this set contains all n people.
- When group of people meet we create a new "meeting vertex". We add edges to this vertex from vertices in which pawns of meeting's participants currently are. Now, we move these pawns to the new vertex. If any of the people in the meeting were in *possibly_infected*, then we need to put all of the participants in this set.

We can represent current state of our knowledge as a directed acyclic graph. Initially, it consists of n isolated vertices, on the i-th of then we put a pawn corresponding to the i-th person.

- We keep a set *possibly_infected* of all people who might be infected and are not under quarantine. Initially, this set contains all n people.
- When group of people meet we create a new "meeting vertex". We add edges to this vertex from vertices in which pawns of meeting's participants currently are. Now, we move these pawns to the new vertex. If any of the people in the meeting were in *possibly_infected*, then we need to put all of the participants in this set.
- If some person receives positive test result, then we erase this person from *possibly_infected* and remove the corresponding pawn from the graph.

We can represent current state of our knowledge as a directed acyclic graph. Initially, it consists of n isolated vertices, on the i-th of then we put a pawn corresponding to the i-th person.

- We keep a set *possibly_infected* of all people who might be infected and are not under quarantine. Initially, this set contains all n people.
- When group of people meet we create a new "meeting vertex". We add edges to this vertex from vertices in which pawns of meeting's participants currently are. Now, we move these pawns to the new vertex. If any of the people in the meeting were in *possibly_infected*, then we need to put all of the participants in this set.
- If some person receives positive test result, then we erase this person from *possibly_infected* and remove the corresponding pawn from the graph.
- If some person receives negative test result, then we need to update the state of our knowledge.

When person p receives negative test result, then we traverse our graph with DFS, starting from a vertex v in which p's pawn currently stands.

Step of **DFS(v)** algorithm:

- We know that there is a healthy person who took part in the meeting v. Therefore, all participants of this meeting were healthy when they met. For every pawn that is still in v we can deduce that corresponding person is healthy and erase them from *possibly_infected*[1].
- We erase vertex v from the graph.

[1] The rest of the participants must have attended other meetings since v took place, so we cannot be sure that they didn't get infected

Step of **DFS(v)** algorithm:

- We know that there is a healthy person who took part in the meeting v. Therefore, all participants of this meeting were healthy when they met. For every pawn that is still in v we can deduce that corresponding person is healthy and erase them from *possibly_infected*[1].
- We erase vertex v from the graph.
- For every edge $u \to v$, we call **DFS(u)**. We can do it because all participants of v were healthy, so all people that they met must have been healthy too.

[1] The rest of the participants must have attended other meetings since v took place, so we cannot be sure that they didn't get infected

Step of **DFS(v)** algorithm:

- We know that there is a healthy person who took part in the meeting v. Therefore, all participants of this meeting were healthy when they met. For every pawn that is still in v we can deduce that corresponding person is healthy and erase them from *possibly_infected*[1].
- We erase vertex v from the graph.
- For every edge $u \to v$, we call **DFS(u)**. We can do it because all participants of v were healthy, so all people that they met must have been healthy too.
- For every edge $v \to w$, we **erase** such edge from the graph and check if it was **the last** going into w from a meeting with potentially infected people. If so, then we call **DFS(w)**.

[1] The rest of the participants must have attended other meetings since v took place, so we cannot be sure that they didn't get infected

When we receive a query, then we can search for an answer in *possibly_infected*.

If this set is empty, then nobody (but people under the quarantine) is infected and the epidemic has been contained.

Complexity analysis

Let n be the number of people , s – number of meetings and c – sum of meeting sizes.

Complexity analysis

Let n be the number of people , s – number of meetings and c – sum of meeting sizes.

- Our graph has $n + s$ vertices (one initial vertex for every person and one created for every meeting) and (at most) c edges.

Complexity analysis

Let n be the number of people , s – number of meetings and c – sum of meeting sizes.

- Our graph has $n + s$ vertices (one initial vertex for every person and one created for every meeting) and (at most) c edges.
- All DFS calls in all iterations take $\mathcal{O}(n + s + c)$ operations because every visited vertex (or edge) is permanently erased after processing it.

Complexity analysis

Let n be the number of people , s – number of meetings and c – sum of meeting sizes.

- Our graph has $n + s$ vertices (one initial vertex for every person and one created for every meeting) and (at most) c edges.
- All DFS calls in all iterations take $\mathcal{O}(n + s + c)$ operations because every visited vertex (or edge) is permanently erased after processing it.
- Number of operations on *possibly_infected* set is linear with respect to the input size. Every person can be put to this set at most as many times as there were meetings that this person attended (plus one time during initialization). The same goes for the number of times one person can be erased from *possibly_infected*. Every operation on the set takes $\mathcal{O}(\log n)$ time.

THEORETICAL COMPUTER SCIENCE Jagiellonian University

Problem B
Babushka and her pierogi

Author: Daniel Goc

Task

You are given a sequence of n integers a_i and an integer sequence p_i of the same length. In both of these sequences elements are pairwise distinct and consist of the same numbers (p is permutation of a). You are also given number C.

In one move you can choose two indices i, j and swap a_i, a_j paying $|a_i - a_j| + C$.

Your task is to transform a into p at the lowest possible cost.

We can get a lower bound on the cost:
$\frac{1}{2}\sum_{i=1}^{n}|a_i - p_i| + (n - number\_of\_permutation\_cycles) \cdot C$

The second part of the sum comes from the fact that after all operations we have $a = p$, so there are n cycles and a single swap can add at most one cycle, therefore we need at least $n - number\_of\_permutation\_cycles$ operations.

We can get a lower bound on the cost:
$\frac{1}{2}\sum_{i=1}^{n}|a_i - p_i| + (n - number\_of\_permutation\_cycles) \cdot C$

The second part of the sum comes from the fact that after all operations we have $a = p$, so there are n cycles and a single swap can add at most one cycle, therefore we need at least $n - number\_of\_permutation\_cycles$ operations.

It turns out that there exist an algorithm that solves this problem at exactly that cost.

We can solve the problem for each permutation cycle separately, so we will focus on a single cycle.

We can solve the problem for each permutation cycle separately, so we will focus on a single cycle.

Our goal is to find two elements in the cycle, such that after swapping them we have split the cycle into two cycles, and we are still able to achieve the optimal cost. First condition is true for every two distinct elements of the cycle. The second is true in a following situation: let i, j be indices of cycle elements in the sequence, then:

$$a_j \in [min(a_i, p_i), max(a_i, p_i)] \wedge a_i \in [min(a_j, p_j), max(a_j, p_j)]$$

We can solve the problem for each permutation cycle separately, so we will focus on a single cycle.

Our goal is to find two elements in the cycle, such that after swapping them we have split the cycle into two cycles, and we are still able to achieve the optimal cost. First condition is true for every two distinct elements of the cycle. The second is true in a following situation: let i, j be indices of cycle elements in the sequence, then:

$$a_j \in [min(a_i, p_i), max(a_i, p_i)] \wedge a_i \in [min(a_j, p_j), max(a_j, p_j)]$$

We choose element of the cycle such that corresponding element in p is the largest possible. Then there always exists a different element of the cycle such that paired with our chosen element it fulfils the aforementioned condition (it is easy too see that otherwise it would be a cycle with a single element and we wouldn't need to split it).

We can solve the problem for each permutation cycle separately, so we will focus on a single cycle.

Our goal is to find two elements in the cycle, such that after swapping them we have split the cycle into two cycles, and we are still able to achieve the optimal cost. First condition is true for every two distinct elements of the cycle. The second is true in a following situation: let i, j be indices of cycle elements in the sequence, then:

$$a_j \in [min(a_i, p_i), max(a_i, p_i)] \wedge a_i \in [min(a_j, p_j), max(a_j, p_j)]$$

We choose element of the cycle such that corresponding element in p is the largest possible. Then there always exists a different element of the cycle such that paired with our chosen element it fulfils the aforementioned condition (it is easy too see that otherwise it would be a cycle with a single element and we wouldn't need to split it).

If so, then we can search for such an element in our cycle, split it, update sequence a and proceed recursively with two smaller cycles.

Naive search through the whole cycle is too slow, but we can alternate between searching from the beginning and the end of the cycle, which assures that we will check number of elements that is proportional to the size of the smaller of two cycles obtained from the split. This gives $\mathcal{O}(s \log s)$ operations where s is the size of the initial cycle.

Naive search through the whole cycle is too slow, but we can alternate between searching from the beginning and the end of the cycle, which assures that we will check number of elements that is proportional to the size of the smaller of two cycles obtained from the split. This gives $\mathcal{O}(s \log s)$ operations where s is the size of the initial cycle.

We need to keep the cycle in a structure that will allow us to quickly find the largest element and to add/erase elements. A standard set is a good option which gives us overall complexity of $\mathcal{O}(n \log^2 n)$. It is possible to achieve $\mathcal{O}(n \log n)$ using other structures, but it was not required.

Problem M
Median

Author: Krzysztof Maziarz

Task

We have an incorrect algorithm for computing the median of a sequence. It works in a following way: if the sequence has at most 2 elements then return the correct answer, otherwise split the sequence into 3 parts with equal length, recursively compute the answer for them and then return median of these 3 values.

Task

We have an incorrect algorithm for computing the median of a sequence. It works in a following way: if the sequence has at most 2 elements then return the correct answer, otherwise split the sequence into 3 parts with equal length, recursively compute the answer for them and then return median of these 3 values.

Given sequence of n integers in range $[0, m-1]$, with q unknown numbers, compute (modulo $10^9 + 7$) in how many ways can we choose the unknown values (from range $[0, m-1]$) in such a way that the algorithm returns the correct median of a sequence.

Task

We have an incorrect algorithm for computing the median of a sequence. It works in a following way: if the sequence has at most 2 elements then return the correct answer, otherwise split the sequence into 3 parts with equal length, recursively compute the answer for them and then return median of these 3 values.

Given sequence of n integers in range $[0, m-1]$, with q unknown numbers, compute (modulo 10^9+7) in how many ways can we choose the unknown values (from range $[0, m-1]$) in such a way that the algorithm returns the correct median of a sequence.

Limits: $n \leq 3^8 = 6561$, $q \leq 30$, $m \leq 10^9$.

Let's fix $t \in [0, m-1]$. In how many ways can we choose the unknown values such that both the correct median and the one returned by the algorithm equals t?

Let's fix $t \in [0, m-1]$. In how many ways can we choose the unknown values such that both the correct median and the one returned by the algorithm equals t?

Let's replace the numbers smaller than t with -1, the equal ones with 0 and the bigger ones with 1. Let's denote the number of -1s and $1s$ that replaced the unknown numbers by x and y respectively ($x, y \in [0, q]$). It is easy to verify if the correct median is 0 knowing x, y.

Let's fix $t \in [0, m-1]$. In how many ways can we choose the unknown values such that both the correct median and the one returned by the algorithm equals t?

Let's replace the numbers smaller than t with -1, the equal ones with 0 and the bigger ones with 1. Let's denote the number of -1s and $1s$ that replaced the unknown numbers by x and y respectively ($x, y \in [0, q]$). It is easy to verify if the correct median is 0 knowing x, y.

We will compute in how many ways can we choose the unknown values (from set $\{-1, 0, 1\}$) in such a way that the algorithm returns 0.

It can be done for all (x, y) at once using dynamic programming on a tree of the recurrence. In every node we keep the result for every tuple: (x, y, a) where $a \in \{-1, 0, 1\}$ is an answer for the node.

It can be done for all (x, y) at once using dynamic programming on a tree of the recurrence. In every node we keep the result for every tuple: (x, y, a) where $a \in \{-1, 0, 1\}$ is an answer for the node.

If for given x i y the correct median is 0 we can count the number of solutions by multiplying the result from dp by $t^x(m-1-t)^y$.

It can be done for all (x, y) at once using dynamic programming on a tree of the recurrence. In every node we keep the result for every tuple: (x, y, a) where $a \in \{-1, 0, 1\}$ is an answer for the node.

If for given x i y the correct median is 0 we can count the number of solutions by multiplying the result from dp by $t^x(m-1-t)^y$.

The dynamic solution is $\mathcal{O}(n+q^4)$, so we can't run it for every t.

It can be done for all (x, y) at once using dynamic programming on a tree of the recurrence. In every node we keep the result for every tuple: (x, y, a) where $a \in \{-1, 0, 1\}$ is an answer for the node.

If for given x i y the correct median is 0 we can count the number of solutions by multiplying the result from dp by $t^x(m - 1 - t)^y$.

The dynamic solution is $\mathcal{O}(n + q^4)$, so we can't run it for every t.

Let $X = \{x_1 < x_2 < ... < x_l\}$ be the *set* of known values in the sequence .

It can be done for all (x, y) at once using dynamic programming on a tree of the recurrence. In every node we keep the result for every tuple: (x, y, a) where $a \in \{-1, 0, 1\}$ is an answer for the node.

If for given x i y the correct median is 0 we can count the number of solutions by multiplying the result from dp by $t^x(m-1-t)^y$.

The dynamic solution is $\mathcal{O}(n+q^4)$, so we can't run it for every t.

Let $X = \{x_1 < x_2 < ... < x_l\}$ be the *set* of known values in the sequence .

Let $S = \{[0, x_1 - 1], [x_1, x_1], [x_1 + 1, x_2 - 1], [x_2, x_2], \ldots, [x_l + 1, m - 1]\}$, $|S| \in \mathcal{O}(n)$. If $p \in S$ the for each $t \in p$ the dynamic solution returns the exact same results.

One can prove that only $\mathcal{O}(q)$ of the ranges contributes to the final answer. Informal argument: the median of all values won't diverge much from the median of all **known** values.

One can prove that only $\mathcal{O}(q)$ of the ranges contributes to the final answer. Informal argument: the median of all values won't diverge much from the median of all **known** values.

Using that fact we can run the dynamic solution only $\mathcal{O}(q)$ times so the overall complexity is $\mathcal{O}(nq + q^5)$.

One can prove that only $\mathcal{O}(q)$ of the ranges contributes to the final answer. Informal argument: the median of all values won't diverge much from the median of all **known** values.

Using that fact we can run the dynamic solution only $\mathcal{O}(q)$ times so the overall complexity is $\mathcal{O}(nq + q^5)$.

The only problem is that we have to multiply the results from dp by $\sum_{t=L}^{R} t^x(m-1-t)^y$ for many different x, y, L, R.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

We are interested in $Q_{x,y}(R) - Q_{x,y}(L-1)$.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

We are interested in $Q_{x,y}(R) - Q_{x,y}(L-1)$.

Known fact: $Q_{x,y}(t)$ is a polynomial of degree:
$deg(P_{x,y}(t)) + 1 = x + y + 1$.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

We are interested in $Q_{x,y}(R) - Q_{x,y}(L-1)$.

Known fact: $Q_{x,y}(t)$ is a polynomial of degree:
$deg(P_{x,y}(t)) + 1 = x + y + 1$.

Proof: Faulhaber's formula.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

We are interested in $Q_{x,y}(R) - Q_{x,y}(L-1)$.

Known fact: $Q_{x,y}(t)$ is a polynomial of degree:
$deg(P_{x,y}(t)) + 1 = x + y + 1$.

Proof: Faulhaber's formula.

We can compute it's values in first $x + y + 2$ points:
$Q_{x,y}(0), Q_{x,y}(1), \ldots Q_{x,y}(x + y + 1)$.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

We are interested in $Q_{x,y}(R) - Q_{x,y}(L-1)$.

Known fact: $Q_{x,y}(t)$ is a polynomial of degree:
$deg(P_{x,y}(t)) + 1 = x + y + 1$.

Proof: Faulhaber's formula.

We can compute it's values in first $x + y + 2$ points:
$Q_{x,y}(0), Q_{x,y}(1), \ldots Q_{x,y}(x+y+1)$.

Then use polynomial interpolation directly or indirectly to compute $Q_{x,y}(T)$ for any given T in $\mathcal{O}(q)$.

Let $P_{x,y}(t) = t^x(m-1-t)^y$.
Let $Q_{x,y}(t) = \sum_{i=0}^{t} P_{x,y}(i)$.

We are interested in $Q_{x,y}(R) - Q_{x,y}(L-1)$.

Known fact: $Q_{x,y}(t)$ is a polynomial of degree:
$deg(P_{x,y}(t)) + 1 = x + y + 1$.

Proof: Faulhaber's formula.

We can compute it's values in first $x+y+2$ points:
$Q_{x,y}(0), Q_{x,y}(1), \dots Q_{x,y}(x+y+1)$.

Then use polynomial interpolation directly or indirectly to compute $Q_{x,y}(T)$ for any given T in $\mathcal{O}(q)$.

Total complexity $\mathcal{O}(nq + q^5)$.

Jury

Lech Duraj
Krzysztof Maziarz
Krzysztof Kleiner
Daniel Goc
Mateusz Radecki

Beta testers

Rafał Burczyński
Marcin Briański
Kamil Rajtar
Witold Jarnicki
Jan Tułowiecki
Adam Szady
Mateusz Radecki
Kamil Dębowski
Marek Sommer

Thank you!