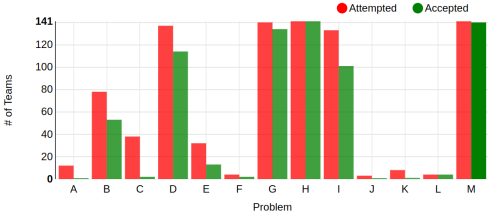
Problem Analysis Session

SWERC judges

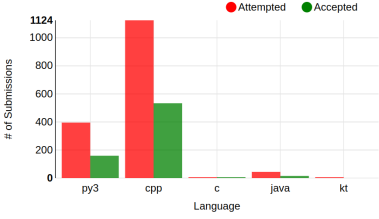
01/12/2024

Statistics

Number of submissions: about 1574

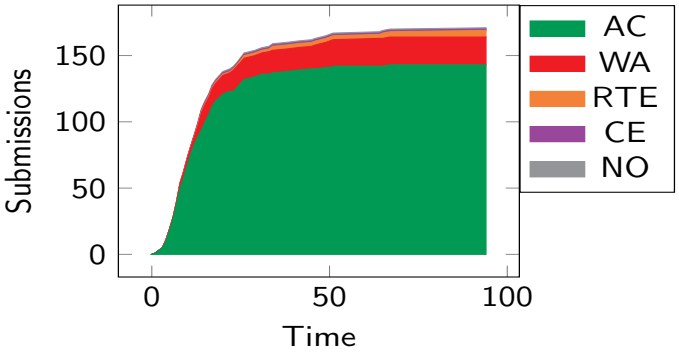


Number of clarification requests: 243 (about 20 answered “No comment.”)



H: The king of SWERC

Solved by all teams before freeze.
First solved after 1 min by the University of Artois team which managed to break our LaTeX slides with their team name.



H: The king of SWERC

Problem

Given a list of names, which one appears the most?

Solution

In a dictionary/hashmap, store for each name the number of times it appears, and increment it when you encounter a name. Then, output the most frequent name.

H: The king of SWERC

Problem

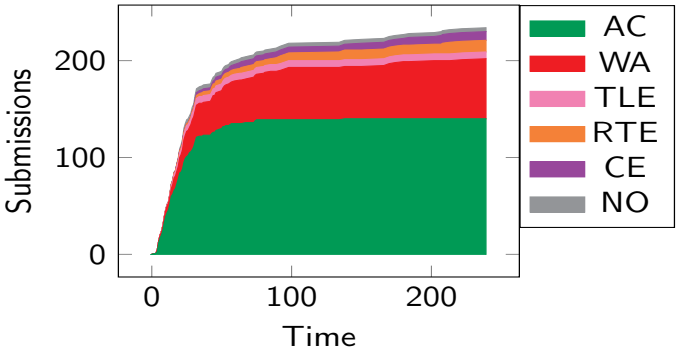
Given a list of names, which one appears the most?

Solution

In a dictionary/hashmap, store for each name the number of times it appears, and increment it when you encounter a name. Then, output the most frequent name.
Actually, doing it naively with arrays is fast enough for the time limits.

M: Ook? Ook!

Solved by 140 teams before freeze.
First solved after 3 min by ensipc1.



M: Ook? Ook!

Problem

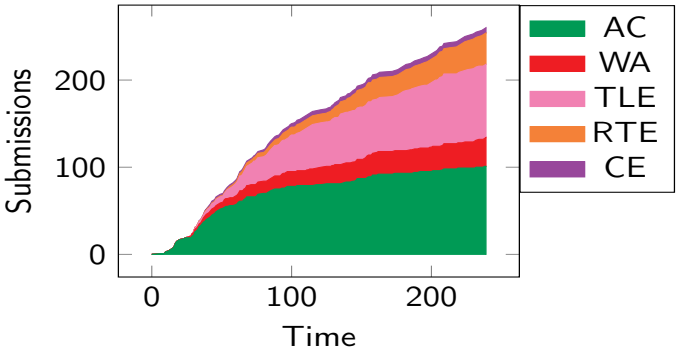
You are given the translation of `OK`, `KO` and `OOK` in a Morse-like language. Can you translate any word which uses only `O` and `K`?

Solution

1. The word `OOK` translates to a word of size 10, and the word `OK` to a word of size 4, so the first four letters of `OK` correspond to the translation of `O`: `.-.-`
2. Thus, the last remaining letters of `OK` correspond to letter `K`: `.-`
3. Given any word consisting of only `O` and `K`, we can then concatenate the translations of `O` and `K`

I: Divination

Solved by 101 teams before freeze.
First solved after 9 min by Schwebler's Peacocks.



I: Divination

Problem

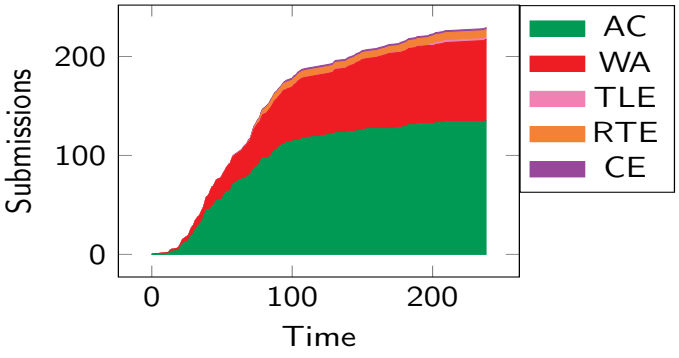
Determine whether there exists a path in a directed acyclic graph that includes all vertices.

Solution – $O(V + E)$ time

- This problem is equivalent to: whether the topological order is unique.
  - If there exists such a path, obviously the topological order is unique.
  - If the topological order is unique, there must be an edge between every pair of adjacent vertices in the order. Otherwise they can be exchanged, and consequently the topological order is not unique.
- We can use a traditional topological sorting algorithm. In each step, if there are multiple vertices having in-degree 0, return No. Otherwise, when the topological sorting finishes, return Yes.

G: Guess How the Ballet Will End

Solved by 134 teams before freeze.
First solved after 12 min by morETHanusual.



G: Guess How the Ballet Will End

Problem

Saturating addition: define (for $0 \leqslant x \leqslant R$)

$$x \oplus y = \begin{cases} min(x + y, R), & \text{if } y \geqslant 0 \\ max(x + y, 0), & \text{if } y < 0 \end{cases}$$

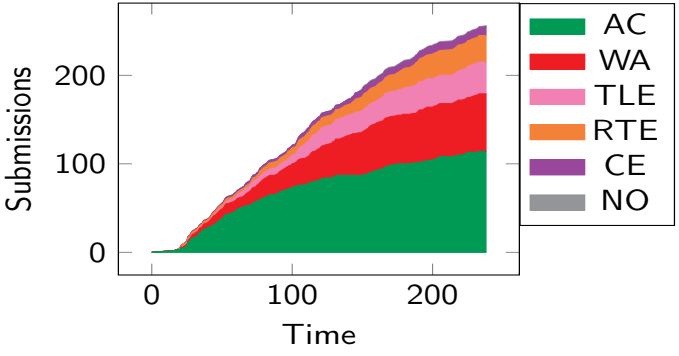
Will all integers in $[0, R]$ have the same value after a given list of saturating additions?

Constant time solution

Perform the saturating additions on 0 and R and check whether they have the same value. If they do, return it.

D: Temple Architecture

Solved by 114 teams before freeze.
First solved after 16 min by UPC-1.



D: Temple Architecture

Problem

For each element in a list H, find the nearest greater element.

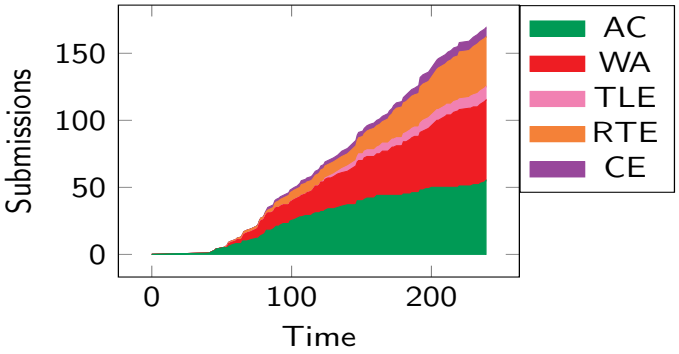
Solution – $O(N)$ time

Use *monotonic (decreasing) queues* to compute the closest greater element on the left/right.

- Going left to right, and using an array *left*, initially set to -1, to store the position of the closest greater element on the left:
  - for each $H(i)$, pop the queue until either the queue is empty or a greater element is found.
  - set *left(i)* to the value at the end of the queue, if there is one. This is the closest greater element of the left for $H(i)$.
  - add i to the end of the queue.
- Going right to left, repeat this process for the closest greater element on the *right*.
- In the end, for each i, choose the closest greater element between $left(i)$ and $right(i)$ ($min(i - left(i), right(i) - i)$), and compute the sum.

B: Divine Gifting

Solved by 54 teams before freeze.
First solved after 41 min by UPC-1.



B: Divine Gifting

Problem

Given target gift delivery days $(d_i)_{i=1,\ldots,N}$, find actual delivery days $(a_i)_{i=1,\ldots,N}$ that minimize $\sum_{i=1}^{N} (d_i - a_i)^2$ with the constraints:

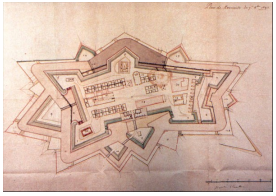
- there are at most K different a_i's;
- $a_i \geqslant d_i$ for $i = 1, \ldots, N$.

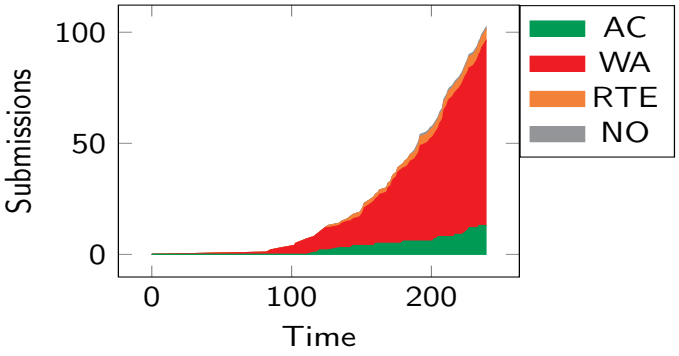
Solution – $O(N^2 * K)$ time

Dynamic programming on state (current gift, number of actual delivery days left) of cardinality $N \times K$. For each state, try all possibilities for the next delivery day (there are $O(N)$ of them). It was needed to incrementally compute sums $\sum (d_j - A)^2$ as their upper index and A increased, in order not to pay an additional N factor in time complexity.

E: Building the Fort

Solved by 13 teams before freeze.
First solved after 116 min by flag[10].





E: Building the Fort

Problem

Construct a simple polygon with N fixed vertices and at most $3N$ total vertices with no interior points.

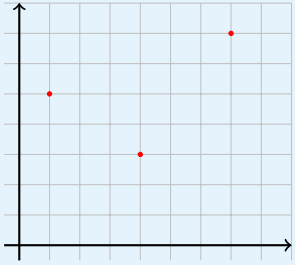
Solution

E: Building the Fort

Problem

Construct a simple polygon with N fixed vertices and at most $3N$ total vertices with no interior points.

Solution

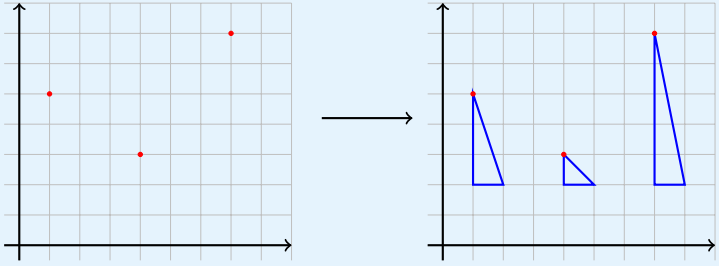


E: Building the Fort

Problem

Construct a simple polygon with N fixed vertices and at most $3N$ total vertices with no interior points.

Solution

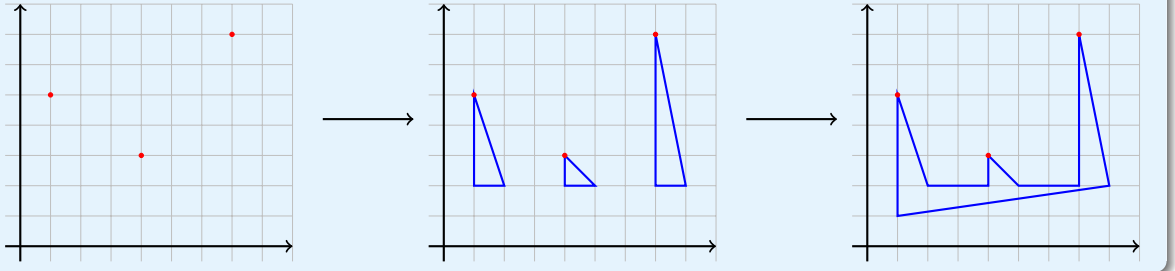


E: Building the Fort

Problem

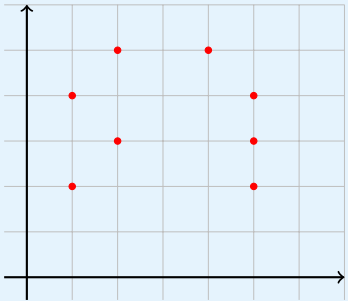
Construct a simple polygon with N fixed vertices and at most $3N$ total vertices with no interior points.

Solution



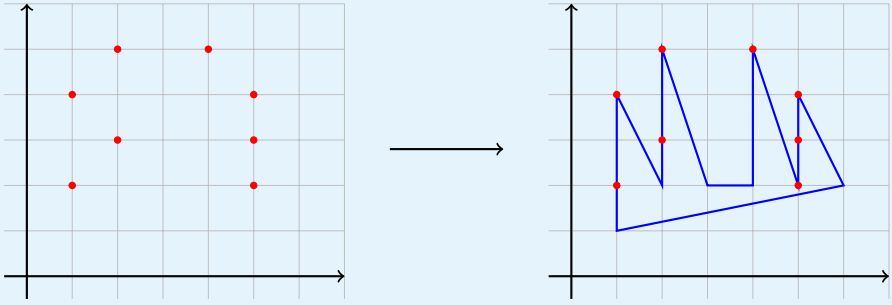
E: Building the Fort

Solution



E: Building the Fort

Solution



E: Building the Fort

Solution

E: Building the Fort

Solution

- Group the vertices by their x-coordinates. For each group in ascending x-order:

E: Building the Fort

Solution

- Group the vertices by their x-coordinates. For each group in ascending x-order:
  - add a point at $(x, 2)$;

E: Building the Fort

Solution

- Group the vertices by their x-coordinates. For each group in ascending x-order:
  - add a point at $(x, 2)$;
  - add the vertices in ascending y-order;

E: Building the Fort

Solution

- Group the vertices by their x-coordinates. For each group in ascending x-order:
  - add a point at $(x, 2)$;
  - add the vertices in ascending y-order;
  - add a point at $(x + 1, 2)$.

E: Building the Fort

Solution

- Group the vertices by their x-coordinates. For each group in ascending x-order:
  - add a point at $(x, 2)$;
  - add the vertices in ascending y-order;
  - add a point at $(x + 1, 2)$.
- Close the polygon.

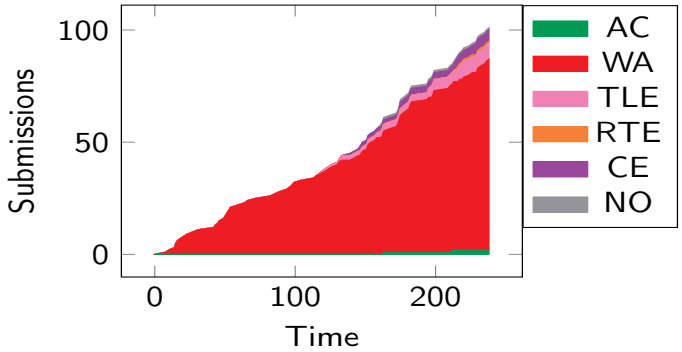
E: Building the Fort

Solution

- Group the vertices by their x-coordinates. For each group in ascending x-order:
  - add a point at $(x, 2)$;
  - add the vertices in ascending y-order;
  - add a point at $(x + 1, 2)$.
- Close the polygon.
- Handle edge cases.

C: Phryctoria

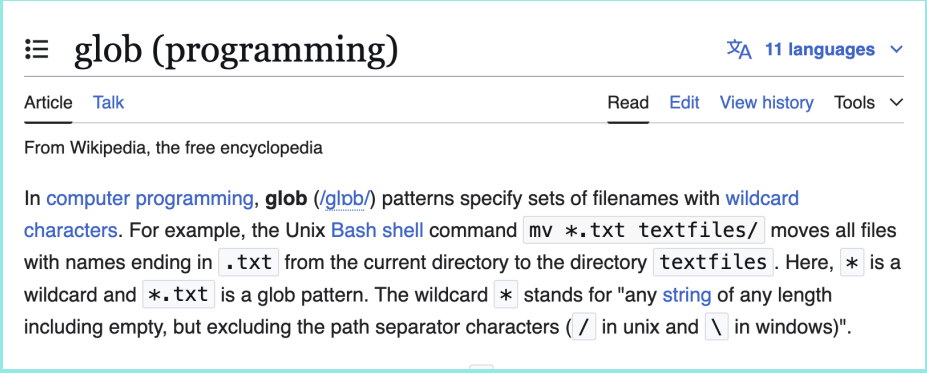
Solved by 2 teams before freeze.
First solved after 163 min by forETHought.



C: Phryctoria

Problem

Find the shortest glob that matches a string S but does not match a string T.



glob (programming)

11 languages

Article Talk Read Edit View history Tools

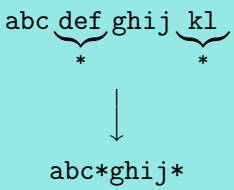
From Wikipedia, the free encyclopedia

In computer programming, **glob** (/glɒb/) patterns specify sets of filenames with wildcard characters. For example, the Unix Bash shell command `mv *.txt textfiles/` moves all files with names ending in `.txt` from the current directory to the directory `textfiles`. Here, `*` is a wildcard and `*.txt` is a glob pattern. The wildcard `*` stands for "any string of any length including empty, but excluding the path separator characters (`/` in unix and `\` in windows)".

C: Phryctoria

Solution

The shortest glob will be S, possibly with some substrings replaced by wildcards.



C: Phryctoria

Solution

We can solve this problem using dynamic programming.

C: Phryctoria

Solution

We can solve this problem using dynamic programming.
Let $\mathrm{dp}[i][j][g]$ be the shortest glob such that:

- the glob matches the first i characters of S;
- the glob matches the first j characters of T, but does **not** match any smaller prefix of T;
- the glob ends with a wildcard iff $g = 1$.

C: Phryctoria

Solution

We have two possible transitions:

- add a wildcard: $\mathrm{dp}[i+k][j][1] = \mathrm{dp}[i][j][0] + 1$, for $k \geqslant 0$;
- add a substring $S[i \ldots i+k]$: find the first position p where the substring occurs in $T[j \ldots]$,
  - if substring is not found, then we have a possible solution;
  - otherwise we can update $\mathrm{dp}[i+k][p+k][0] = \mathrm{dp}[i][j][1] + k$, for $k \geqslant 0$.

C: Phryctoria

Solution

We have two possible transitions:

- add a wildcard: $\mathrm{dp}[i+k][j][1] = \mathrm{dp}[i][j][0] + 1$, for $k \geqslant 0$;
- add a substring $S[i \ldots i+k]$: find the first position p where the substring occurs in $T[j \ldots]$,
  - if substring is not found, then we have a possible solution;
  - otherwise we can update $\mathrm{dp}[i+k][p+k][0] = \mathrm{dp}[i][j][1] + k$, for $k \geqslant 0$.

We can use Z-algorithm to find the first occurrence of $S[i \ldots i+k]$ for each k in $O(n)$ total time.

C: Phryctoria

Solution

We have two possible transitions:

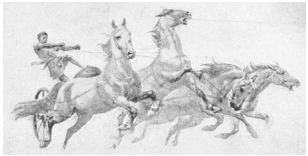
- add a wildcard: $\mathrm{dp}[i+k][j][1] = \mathrm{dp}[i][j][0] + 1$, for $k \geqslant 0$;
- add a substring $S[i \ldots i+k]$: find the first position p where the substring occurs in $T[j \ldots]$,
  - if substring is not found, then we have a possible solution;
  - otherwise we can update $\mathrm{dp}[i+k][p+k][0] = \mathrm{dp}[i][j][1] + k$, for $k \geqslant 0$.

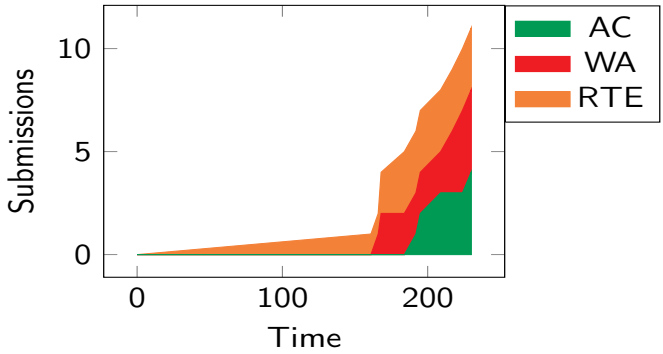
We can use Z-algorithm to find the first occurrence of $S[i \ldots i+k]$ for each k in $O(n)$ total time.

Since we have $O(n^2)$ states and each state can be updated in $O(n)$ time, the total time complexity is $O(n^3)$.

L: The Charioteer

Solved by 4 teams before freeze.
First solved after 192 min by flag[10].





L: The Charioteer

Problem

You can steer Phaeton's chariot, whose speed increases by 1 every second, but you have only 20000 moves.

Determining where the temple is

Let's denote the maximal absolute value of the temple coordinates by M.
There are many possible solutions.
Start by knowing the x coordinate.
First, go front two times. Then, you will know if the x coordinate is $< 2, 2$ or > 2.

L: The Charioteer

Determining where the temple is

- If $x > 2$, then go front until you start distancing yourself from the temple. When that happens, you will know the temple's x coordinate.
- If $x < 2$, then turn left two times and go back, repeating the same strategy of continuing until you find the temple.

At worst, this takes about $O(sqrt(M))$ moves.
Since we know the x coordinate, we can easily know the y coordinate of the temple by moving vertically for one turn.

L: The Charioteer

Going to the temple

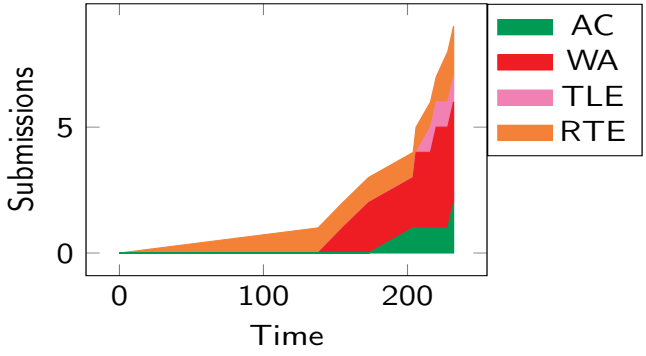
Notice that some sequence of moves produce specific effects which don't depend on the current speed:

- FRRRLRRR will make you face right, while preserving your position
- RRRRLRRR will make you face back, while preserving your position
- RLLLLRRR will make you advance 4 units to the left
- LRRRRLLL will make you advance 4 units to the right

This means that we can always go to the temple in $O(sqrt(M))$ moves, provided that the position mod 4 is correct.
We can use the sequence LFRRRFLRRR which goes 4 units back and 1 unit left.
This means we can change the position to the correct value modulo 4.
Thus, we can reach the temple in $O(sqrt(M))$, which is enough for this problem.

F: Yaxchilán Maze

Solved by 2 teams before freeze.
First solved after 204 min by UPC-1.



F: Yaxchilán Maze

Problem

Find paths to E exits in a N-node maze that changes over T timesteps, with booby-traps that trigger when they are in a connected component of size $> K$. Booby-traps release wasps that spread to currently and newly connected graph nodes.

Solution – $O(T \log(N)(\log(T) + E))$ time

- Use a segment tree over the T timesteps. Each segment tree node contains the **list** of graph edges that are active during the corresponding interval of timesteps.
- Traverse the segment tree using a DFS from root (always going left first).
- During traversal, maintain a Union-Find data structure. When entering a segment tree node, add corresponding graph edges to the Union-Find. When leaving a segment tree node, **rollback** added edges.

F: Yaxchilán Maze

Problem

Find paths to E exits in a N-node maze that changes over T timesteps, with booby-traps that trigger when they are in a connected component of size $> K$. Booby-traps release wasps that spread to currently and newly connected graph nodes.

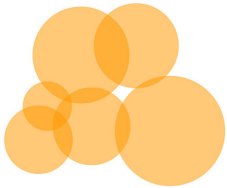
Solution – $O(T \log(N)(\log(T) + E))$ time

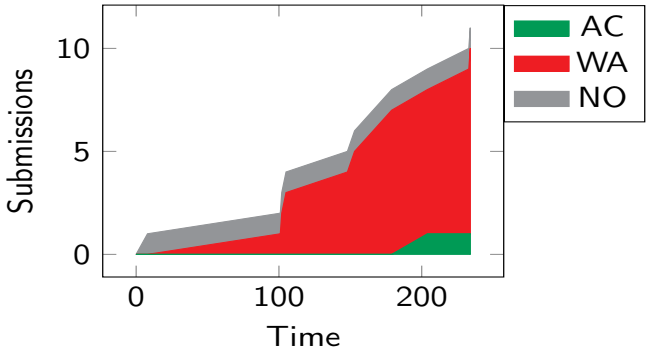
- The Union-Find must maintain connected component sizes, presence of booby-traps, bitset of connectivity to the start graph nodes, presence of wasps.
- When rolling back Union-Find changes, don't rollback the connectivity to start nodes bitset, nor the presence of wasps. Instead, if applicable, propagate them to both newly connected components formed by rolling back a connection.

The Union-Find data structure does not use path compression, as it adds work during the rollback phase. It uses only the union-by-rank optimization, hence the $\log(N)$ factor.

K: Disk Covering

Solved by 1 teams before freeze.
First solved after 204 min by BaguETHe.





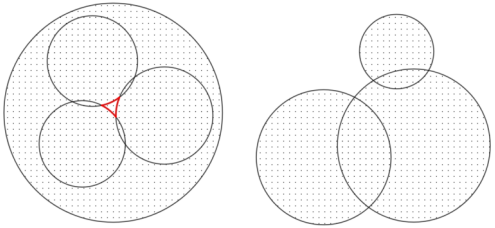
K: Disk Covering

Problem

Determine if there exists a place completely surrounded by disks, yet not on the disks.

Preprocessing

- If two disks intersect, they're regarded connected. Calculate the connected components. In the following solutions, we only elaborate on the solution for one connected component.
- To avoid the following case, remove all disks that are fully in other disks in advance.



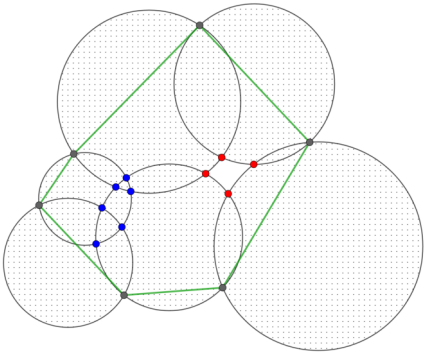
K: Disk Covering

Problem

Determine if there exists a place completely surrounded by disks, yet not on the disks.

Solution 1 (incorrect)

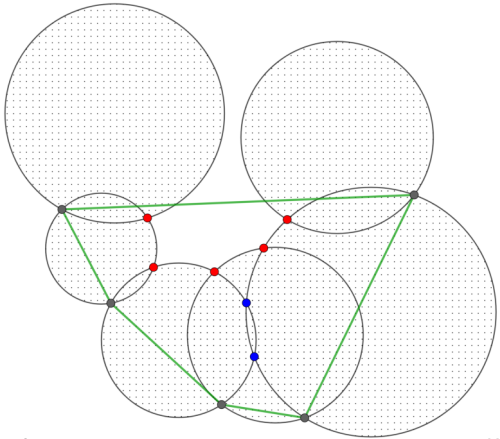
- We find all the intersection points of any two disks. Calculate their convex hull.
- If there's any intersection inside the convex hull & not inside any other disk (other than the two disks that form it), output Yes and exit.
- Otherwise output No.



K: Disk Covering

Solution 1 (incorrect)

Counterexample of this solution.



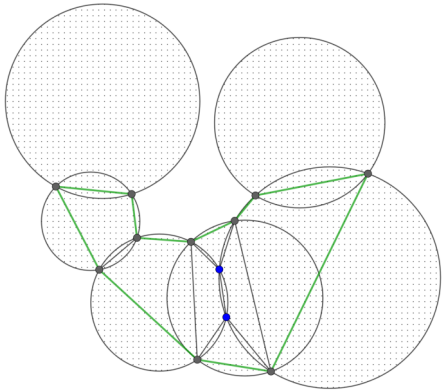
K: Disk Covering

Problem

Determine if there exists a place completely surrounded by disks, yet not on the disks.

Solution 2 (incorrect)

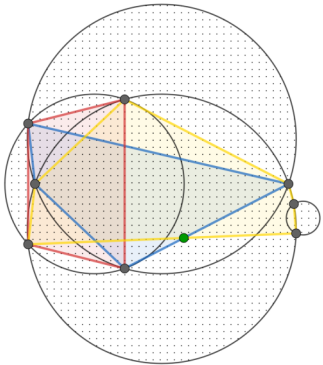
- For each circle we draw a polygon that is constructed by all the intersections on it. Given the nature of these polygons, the vertices of the outline of their union are all existing intersections.
- To get the outline, start at the northernmost intersection. Then we walk through the boundary clockwise.
- Same as above, check if there's any intersection not on the outline & not inside any other disk.



K: Disk Covering

Solution 2 (incorrect)

Counterexample of this solution. If we merge the 3 polygons, the green point is a vertex of the outline but not an existing intersection.



K: Disk Covering

Problem

Determine if there exists a place completely surrounded by disks, yet not on the disks.

Solution 3

- Based on the previous solution, we can regard the polygons as arbitrary ones, and use some polygon union algorithms (brute force, sweep line, etc.) to calculate the outline.
- However, either they require an overwhelming coding complexity, or their worst case time complexity is above $O(N^4)$, or even both.

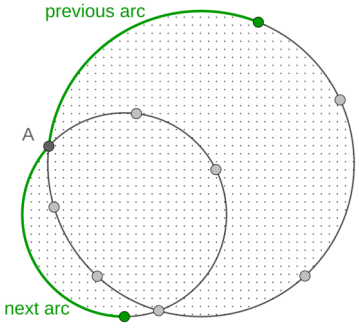
K: Disk Covering

Problem

Determine if there exists a place completely surrounded by disks, yet not on the disks.

Solution 4 - $O(N^3)$ time

- For each circle, we pre-process all the intersections on this circle, and order them according to their polar angle (relative to the center of that circle).
- For each intersection A, it is formed by two circles. For each of the two circles, among all intersections on that circle, one is A's previous, one is A's next, in terms of counterclockwise direction.
- Then we can find A's previous arc and next arc on the outline of union of these two disks.



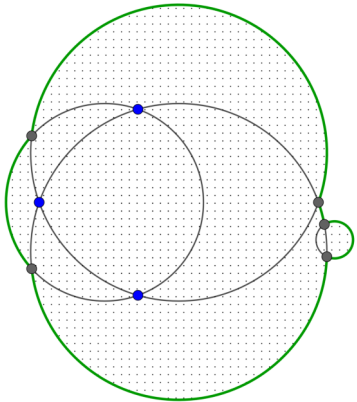
K: Disk Covering

Problem

Determine if there exists a place completely surrounded by disks, yet not on the disks.

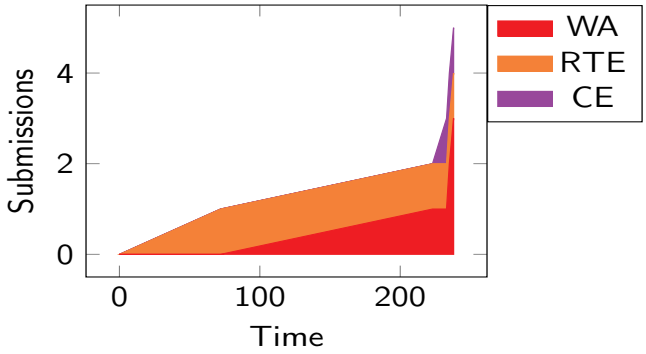
Solution 4 - $O(N^3)$ time

- With the above pre-processing, we can start from the northernmost arc, and do a clockwise or counterclockwise traversal, to form the outline.
- Note that we shouldn't start from the northernmost intersection, as it may not be on the outline.
- Like previous solutions, check if there's any intersection not on the outline & not inside any other disk.



J: Recovering the Tablet

Not solved before freeze.



J: Recovering the Tablet

Problem

Solve a Kakuro with repetitions, while getting as close as possible to the target values.

Verify if it is possible

First, subtract 1 from every white cell, and update the sums accordingly.
Now, all the values are between 0 and 8.
Then, we can create a flow graph to represent the problem.

J: Recovering the Tablet

Verify if it is possible

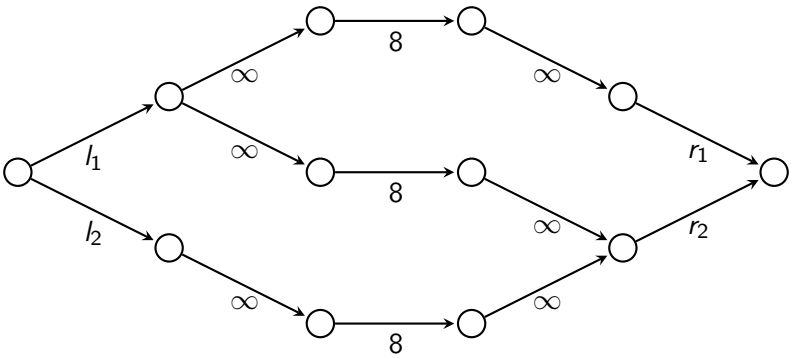
We add a vertex for each horizontal and vertical constraint.
For every white cell, we add two vertices V_{in} and V_{out} and connect them with a directed edge of capacity 8.
Then, connect the horizontal constraint to the corresponding V_{in} vertices with a directed edge of capacity infinity.
Connect the vertices V_{out} with the corresponding vertical constraint vertices with a directed edge of capacity infinity.

J: Recovering the Tablet

Verify if it is possible

Then, connect the source with every horizontal constraint, with an edge of capacity equal to the constraint sum.
Similarly, connect every vertical constraint with the sink with an edge of capacity equal to the constraint sum.
Run max-flow algorithm, and check if the edges starting at source and ending at sink are all full.
However, this does not calculate the optimal solution.

J: Recovering the Tablet



J: Recovering the Tablet

Calculate optimal solution

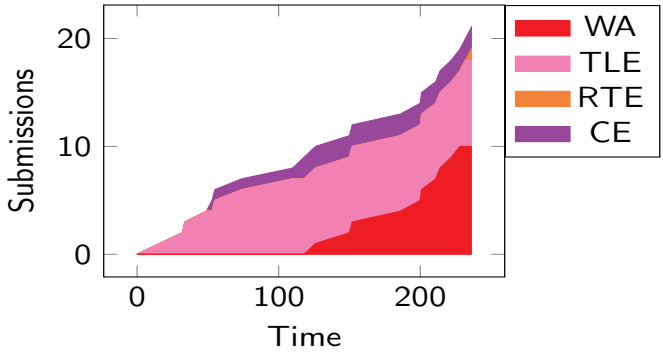
We apply costs to edges.
Given a white cell, if the target is T_i, then we divide the edge between V_{in} and V_{out} in two edges:

- One edge coming from V_{in} to V_{out} with capacity T_i and cost -1.
- One edge coming from V_{in} to V_{out} with capacity $8 - T_i$ and cost 1.

This prioritizes the edge with lower cost for the first T_i units of flow, thus mimicking the absolute value function.
Thus, the final answer is the min cost maximum flow + the sum of all T_i's.

A: Titanomachy

Not solved before freeze.



A - Titanomachy

We are given a more challenging version of the max subarray sum problem, where we can increase/decrease every value by a constant and query in subsegments.

Solution 1 – $O(NQ)$ time

Naive solution

- We can update every segment in $O(N)$
- For every query, use Kadane's algorithm to solve in $O(N)$.

$$DP_i = max(0, DP_{i-1} + A_i), max(DP_i)$$

This won't pass the time limit however.

A - Titanomachy

Notice that the updates are very constrained, and after updating $+A$ and $+B$, it's like updating $+A+B$. Therefore, the upgrades can stack. Let's denote the sum of the upgrades so far as X.

Solution 2 – $O(Nlog(N) + Qlog^2(N))$ time

Let's use a segment tree, where for each node we store information about the sum of the node, the maximum prefix sum, the maximum suffix sum and the maximum sum inside the segment. Suppose that our segment has length S.

Important observation: for every fixed segment, the sum of this segment is a linear function in X.

Solution 2 – $O(Nlog(N) + Qlog^2(N))$ time

Therefore, the sum of the node is a linear function, and the maximum suffix/prefix sum is a piecewise linear convex function, because it is the maximum of a set of linear functions, which we can maintain with a convex hull data structure.

Since the slope of the line is equal to the length of the segment, the number of different lines is S, which means we use $O(Nlog(N))$ memory.

Then, we can calculate the prefix/suffix hulls in $O(S)$, and then merge in $O(S)$ using either two pointers+max convolution or Minkowski sum.

Each update can be processed in $O(1)$, and each query in $O(log^2(N))$, meaning the total time is $O(Nlog(N) + Qlog^2(N))$.