NCPC 2017
Presentation of solutions

The Jury

2017-10-07

NCPC 2017 Jury

- Per Austrin (KTH Royal Institute of Technology)
- Pål Grønås Drange (Statoil ASA)
- Markus Fanebust Dregi (Statoil ASA/Webstep)
- Antti Laaksonen (CSES)
- Ulf Lundström (Excillum)
- Jimmy Mårdell (Spotify)
- Lukáš Poláček (Google)
- Johan Sannemo (Google)
- Pehr Söderman (Kattis)

J — Judging Moose

Problem

Classify moose based on their horns.

Some solution (guess the language)

```
solve(0, 0) :-
    !, write('Not a moose').
solve(L, R) :-
    type(L, R, Type),
    Val is 2*max(L, R),
    write(Type), write(' '), write(Val).
type(L, L, "Even") :- !.
type(_, _, "Odd").
```

Statistics: 347 submissions, 252 accepted, first after 00:03

B — Best Relay Team

Problem

Pick best relay team, given runners' standing and flying start times.

Solution

1. Pre-sort runners by their flying start time

B — Best Relay Team

Problem

Pick best relay team, given runners' standing and flying start times.

Solution

1. Pre-sort runners by their flying start time
2. Try every runner on the first leg

B — Best Relay Team

Problem

Pick best relay team, given runners' standing and flying start times.

Solution

1. Pre-sort runners by their flying start time
2. Try every runner on the first leg
3. For every choice, fill up with 3 fastest remaining flying start runners

B — Best Relay Team

Problem

Pick best relay team, given runners' standing and flying start times.

Solution

1. Pre-sort runners by their flying start time
2. Try every runner on the first leg
3. For every choice, fill up with 3 fastest remaining flying start runners

Complexity is $O(n \log n)$. Many other solutions are also possible.

B — Best Relay Team

Problem

Pick best relay team, given runners' standing and flying start times.

Solution

1. Pre-sort runners by their flying start time
2. Try every runner on the first leg
3. For every choice, fill up with 3 fastest remaining flying start runners

Complexity is $O(n \log n)$. Many other solutions are also possible.

Statistics: 491 submissions, 189 accepted, first after 00:08

G — Galactic Collegiate Programming Contest

Problem

There are n teams who solve m problems in an ICPC style programming contest. After each successful submission, print the rank of your team.

Solution

1. Maintain a set S: the teams whose score is better than your team's score. Your rank is $|S| + 1$.

G — Galactic Collegiate Programming Contest

Problem

There are n teams who solve m problems in an ICPC style programming contest. After each successful submission, print the rank of your team.

Solution

1. Maintain a set S: the teams whose score is better than your team's score. Your rank is $|S| + 1$.
2. When your team solves a problem, remove all teams with a worse score from S.

G — Galactic Collegiate Programming Contest

Problem

There are n teams who solve m problems in an ICPC style programming contest. After each successful submission, print the rank of your team.

Solution

1. Maintain a set S: the teams whose score is better than your team's score. Your rank is $|S| + 1$.
2. When your team solves a problem, remove all teams with a worse score from S.
3. When another team solves a problem, add it to S if its score becomes better than your team's score.

G — Galactic Collegiate Programming Contest

Problem

There are n teams who solve m problems in an ICPC style programming contest. After each successful submission, print the rank of your team.

Solution

1. Maintain a set S: the teams whose score is better than your team's score. Your rank is $|S| + 1$.
2. When your team solves a problem, remove all teams with a worse score from S.
3. When another team solves a problem, add it to S if its score becomes better than your team's score.

The *amortized* complexity of both operations is $O(\log n)$.

G — Galactic Collegiate Programming Contest

Problem

There are n teams who solve m problems in an ICPC style programming contest. After each successful submission, print the rank of your team.

Solution

1. Maintain a set S: the teams whose score is better than your team's score. Your rank is $|S| + 1$.
2. When your team solves a problem, remove all teams with a worse score from S.
3. When another team solves a problem, add it to S if its score becomes better than your team's score.

The *amortized* complexity of both operations is $O(\log n)$.

Statistics: 578 submissions, 79 accepted, first after 00:29

I — Import Spaghetti

Problem

The dependencies form a directed graph, and the task is to find a shortest cycle in a directed graph.

Solution

1. Use the Floyd–Warshall all pairs shortest path algorithm with diagonals initialized to ∞

I — Import Spaghetti

Problem

The dependencies form a directed graph, and the task is to find a shortest cycle in a directed graph.

Solution

1. Use the Floyd–Warshall all pairs shortest path algorithm with diagonals initialized to ∞
2. Afterwards diagonal entry $d(u, u)$ gives length of shortest cycle passing through u.

I — Import Spaghetti

Problem

The dependencies form a directed graph, and the task is to find a shortest cycle in a directed graph.

Solution

1. Use the Floyd–Warshall all pairs shortest path algorithm with diagonals initialized to ∞
2. Afterwards diagonal entry $d(u, u)$ gives length of shortest cycle passing through u.
3. Reconstruct shortest cycle using the distance matrix.

I — Import Spaghetti

Problem

The dependencies form a directed graph, and the task is to find a shortest cycle in a directed graph.

Solution

1. Use the Floyd–Warshall all pairs shortest path algorithm with diagonals initialized to ∞
2. Afterwards diagonal entry $d(u, u)$ gives length of shortest cycle passing through u.
3. Reconstruct shortest cycle using the distance matrix.
4. Alternatively, run BFS from each vertex v to find shortest v–v cycle.

I — Import Spaghetti

Problem

The dependencies form a directed graph, and the task is to find a shortest cycle in a directed graph.

Solution

1. Use the Floyd–Warshall all pairs shortest path algorithm with diagonals initialized to ∞
2. Afterwards diagonal entry $d(u, u)$ gives length of shortest cycle passing through u.
3. Reconstruct shortest cycle using the distance matrix.
4. Alternatively, run BFS from each vertex v to find shortest v–v cycle.

Complexity is $O(n^3)$ or $O(n(n+m))$.

I — Import Spaghetti

Problem

The dependencies form a directed graph, and the task is to find a shortest cycle in a directed graph.

Solution

1. Use the Floyd–Warshall all pairs shortest path algorithm with diagonals initialized to ∞
2. Afterwards diagonal entry $d(u, u)$ gives length of shortest cycle passing through u.
3. Reconstruct shortest cycle using the distance matrix.
4. Alternatively, run BFS from each vertex v to find shortest v–v cycle.

Complexity is $O(n^3)$ or $O(n(n+m))$.

Statistics: 290 submissions, 52 accepted, first after 00:30

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

1. Similar to Dijkstra's or Prim's algorithms:

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

1. Similar to Dijkstra's or Prim's algorithms:
    1. Keep track of *tentative* depth of each square – upper bound on the final water level.

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

1. Similar to Dijkstra's or Prim's algorithms:
    1. Keep track of *tentative* depth of each square – upper bound on the final water level.
    2. At the start, only the drainage point has known depth.

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

1. Similar to Dijkstra's or Prim's algorithms:
    1. Keep track of *tentative* depth of each square – upper bound on the final water level.
    2. At the start, only the drainage point has known depth.
    3. In each iteration, pick the square s with the lowest tentative depth and mark it *final*. Update tentative depth of all neighbours of s.

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

1. Similar to Dijkstra's or Prim's algorithms:
    1. Keep track of *tentative* depth of each square – upper bound on the final water level.
    2. At the start, only the drainage point has known depth.
    3. In each iteration, pick the square s with the lowest tentative depth and mark it *final*. Update tentative depth of all neighbours of s.

Time complexity is $O(n \log n)$, where $n = w \cdot h$ is the size of the grid.

E — Emptying the Baltic

Problem

How much water can we drain from a point at the bottom of the sea?

Solution

1. Similar to Dijkstra's or Prim's algorithms:
    1. Keep track of *tentative* depth of each square – upper bound on the final water level.
    2. At the start, only the drainage point has known depth.
    3. In each iteration, pick the square s with the lowest tentative depth and mark it *final*. Update tentative depth of all neighbours of s.

Time complexity is $O(n \log n)$, where $n = w \cdot h$ is the size of the grid.

Statistics: 296 submissions, 55 accepted, first after 00:47

D — Distinctive Character

Problem

Given n bit vectors of length k, find a bit vector whose minimum Hamming distance is maximum.

Solution

1. There are a total of 2^k possible bit vectors.

D — Distinctive Character

Problem

Given n bit vectors of length k, find a bit vector whose minimum Hamming distance is maximum.

Solution

1. There are a total of 2^k possible bit vectors.
2. Create a graph where each node is a bit vector and there is an edge between two nodes if they differ in a single bit.
   (aka the k-dimensional hypercube graph)

D — Distinctive Character

Problem

Given n bit vectors of length k, find a bit vector whose minimum Hamming distance is maximum.

Solution

1. There are a total of 2^k possible bit vectors.
2. Create a graph where each node is a bit vector and there is an edge between two nodes if they differ in a single bit.
   (aka the k-dimensional hypercube graph)
3. Use a *single* BFS with the n given vectors as sources to find the node whose minimum distance is maximum.

D — Distinctive Character

Problem

Given n bit vectors of length k, find a bit vector whose minimum Hamming distance is maximum.

Solution

1. There are a total of 2^k possible bit vectors.
2. Create a graph where each node is a bit vector and there is an edge between two nodes if they differ in a single bit.
   (aka the k-dimensional hypercube graph)
3. Use a *single* BFS with the n given vectors as sources to find the node whose minimum distance is maximum.

Time complexity is$O(n + k \cdot 2^k)$

D — Distinctive Character

Problem

Given n bit vectors of length k, find a bit vector whose minimum Hamming distance is maximum.

Solution

1. There are a total of 2^k possible bit vectors.
2. Create a graph where each node is a bit vector and there is an edge between two nodes if they differ in a single bit. (aka the k-dimensional hypercube graph)
3. Use a *single* BFS with the n given vectors as sources to find the node whose minimum distance is maximum.

Time complexity is$O(n + k \cdot 2^k)$

Statistics: 461 submissions, 40 accepted, first after 00:17

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

1. When card is sold, at most 6 other cards (the 2 "adjacent" cards of each color) can change their uniqueness values.

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

1. When card is sold, at most 6 other cards (the 2 "adjacent" cards of each color) can change their uniqueness values.
2. For $c \in \{R, G, B\}$, keep set S_c of cards ordered by angle in color c.

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

1. When card is sold, at most 6 other cards (the 2 "adjacent" cards of each color) can change their uniqueness values.
2. For $c \in \{R, G, B\}$, keep set S_c of cards ordered by angle in color c.
3. When selling card, find the ≤ 6 affected cards and recompute their uniqueness values, using fast lookup in the sets S_c.

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

1. When card is sold, at most 6 other cards (the 2 "adjacent" cards of each color) can change their uniqueness values.
2. For $c \in \{R, G, B\}$, keep set S_c of cards ordered by angle in color c.
3. When selling card, find the ≤ 6 affected cards and recompute their uniqueness values, using fast lookup in the sets S_c.
4. Keep all cards in another ordered set ordered by uniqueness value for fast extraction of next card to sell.

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

1. When card is sold, at most 6 other cards (the 2 "adjacent" cards of each color) can change their uniqueness values.
2. For $c \in \{R, G, B\}$, keep set S_c of cards ordered by angle in color c.
3. When selling card, find the ≤ 6 affected cards and recompute their uniqueness values, using fast lookup in the sets S_c.
4. Keep all cards in another ordered set ordered by uniqueness value for fast extraction of next card to sell.

Complexity is $O(n \log n)$ with balanced search trees or similar.

C — Compass Card Sales

Problem

Dynamically keep track of "uniqueness values" of cards while cards are being sold off.

Solution

1. When card is sold, at most 6 other cards (the 2 "adjacent" cards of each color) can change their uniqueness values.
2. For $c \in \{R, G, B\}$, keep set S_c of cards ordered by angle in color c.
3. When selling card, find the ≤ 6 affected cards and recompute their uniqueness values, using fast lookup in the sets S_c.
4. Keep all cards in another ordered set ordered by uniqueness value for fast extraction of next card to sell.

Complexity is $O(n \log n)$ with balanced search trees or similar.

Statistics: 105 submissions, 14 accepted, first after 01:10

K — Kayaking

Problem

Assign pool of weak/normal/strong people to 2-person kayaks (of different speed factors) to maximize speed of slowest kayak.

Solution

K — Kayaking

Problem

Assign pool of weak/normal/strong people to 2-person kayaks (of different speed factors) to maximize speed of slowest kayak.

Solution

1. Binary search over the answer.

K — Kayaking

Problem

Assign pool of weak/normal/strong people to 2-person kayaks (of different speed factors) to maximize speed of slowest kayak.

Solution

1. Binary search over the answer.
2. Check feasibility by greedily assigning people to kayaks
   - kayaks requiring strong+strong or strong+normal get that
   - kayaks that can handle weak+weak or weak+normal get that
   - pair up remaining weaks with strongs and normals with normals and check if this can make all kayaks fast enough

K — Kayaking

Problem

Assign pool of weak/normal/strong people to 2-person kayaks (of different speed factors) to maximize speed of slowest kayak.

Solution

1. Binary search over the answer.
2. Check feasibility by greedily assigning people to kayaks
    - kayaks requiring strong+strong or strong+normal get that
    - kayaks that can handle weak+weak or weak+normal get that
    - pair up remaining weaks with strongs and normals with normals and check if this can make all kayaks fast enough

Time complexity is $O(n \log n)$ for n people.

K — Kayaking

Problem

Assign pool of weak/normal/strong people to 2-person kayaks (of different speed factors) to maximize speed of slowest kayak.

Solution

1. Binary search over the answer.
2. Check feasibility by greedily assigning people to kayaks
   - kayaks requiring strong+strong or strong+normal get that
   - kayaks that can handle weak+weak or weak+normal get that
   - pair up remaining weaks with strongs and normals with normals and check if this can make all kayaks fast enough

Time complexity is $O(n \log n)$ for n people.

Statistics: 82 submissions, 23 accepted, first after 00:46

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution

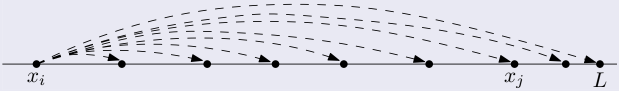
1. Let $S(i)$ be best time if we start from i'th cart.

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



1. Let $S(i)$ be best time if we start from i'th cart.
2. Easy dynamic programming: for each $j > i$, try buying next coffee at cart j

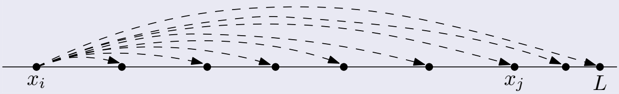
$$S(i) = \min_{j>i} S(j) + \text{Time to go from } x_i \text{ to } x_j$$

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



1. Let $S(i)$ be best time if we start from i'th cart.
2. Easy dynamic programming: for each $j > i$, try buying next coffee at cart j

$$S(i) = \min_{j>i} S(j) + \text{Time to go from } x_i \text{ to } x_j$$

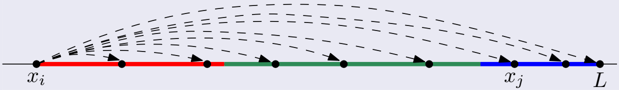
3. Alas, leads to $\Omega(n^2)$ time – too slow!.

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



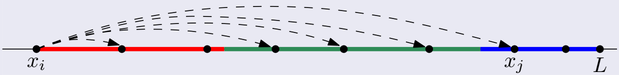
1. From each x_i, three categories of choice for best next cart x_j: During cooldown, during drinking, and after finishing the coffee

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



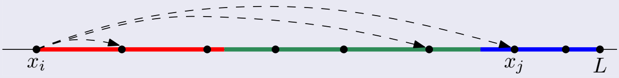
1. From each x_i, three categories of choice for best next cart x_j: During cooldown, during drinking, and after finishing the coffee
2. Before/After drinking: best to pick **smallest** such j (get next coffee as soon as possible)

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



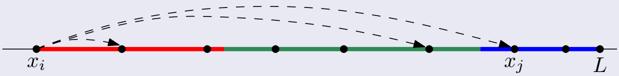
1. From each x_i, three categories of choice for best next cart x_j: During cooldown, during drinking, and after finishing the coffee
2. Before/After drinking: best to pick **smallest** such j (get next coffee as soon as possible)
3. During drinking: best to pick **largest** such j (keep drinking coffee as long as possible)

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



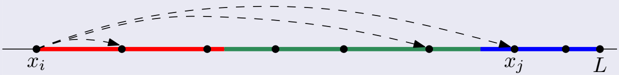
1. Only need to consider three values of j when computing $S(i)$.

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



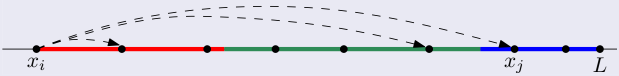
1. Only need to consider three values of j when computing $S(i)$.
2. Can use binary search over cart positions to find them in $O(\log n)$ time.

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



1. Only need to consider three values of j when computing $S(i)$.
2. Can use binary search over cart positions to find them in $O(\log n)$ time.

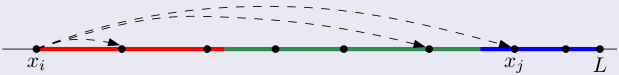
Overall complexity $O(n \log n)$.
(Exercise: can be improved to $O(n)$ time.)

A — Airport Coffee

Problem

Find fastest way of walking distance L. At certain points x_i we can choose to get a future temporary speedup (by buying a coffee), at the cost of cancelling any current speedup.

Solution



1. Only need to consider three values of j when computing $S(i)$.
2. Can use binary search over cart positions to find them in $O(\log n)$ time.

Overall complexity $O(n \log n)$.
(Exercise: can be improved to $O(n)$ time.)

Statistics: 65 submissions, 7 accepted, first after 02:50

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 1

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 1

1. Sort all rays and points by angle.

H — Hubtown

Problem

Given a set of rays from in 2D $(0,0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 1

1. Sort all rays and points by angle.
2. For each point, compare distances to its two neighboring rays (Use sweep approach or binary search to find the two rays quickly)

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 1

1. Sort all rays and points by angle.
2. For each point, compare distances to its two neighboring rays (Use sweep approach or binary search to find the two rays quickly)
3. Caveat! If using doubles, need to be careful with ϵ. (Turns out, distances can differ by $\approx 10^{-13}$ without being equal.)

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 1

1. Sort all rays and points by angle.
2. For each point, compare distances to its two neighboring rays (Use sweep approach or binary search to find the two rays quickly)
3. Caveat! If using doubles, need to be careful with ϵ. (Turns out, distances can differ by $\approx 10^{-13}$ without being equal.)
4. Can also check this using only integer computations. (But, despite small coordinates, need 64 bits.)

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 1

1. Sort all rays and points by angle.
2. For each point, compare distances to its two neighboring rays (Use sweep approach or binary search to find the two rays quickly)
3. Caveat! If using doubles, need to be careful with ϵ. (Turns out, distances can differ by $\approx 10^{-13}$ without being equal.)
4. Can also check this using only integer computations. (But, despite small coordinates, need 64 bits.)
5. Points with a unique neighboring ray can be immediately assigned to that ray (if it has capacity left).

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 1: solve using max flow (merging all points with the same angle into a single node with capacity = #points)

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 1: solve using max flow (merging all points with the same angle into a single node with capacity = #points)
   1. Time complexity with Ford-Fulkerson is $O(p^2)$ where p is the number of train lines adjacent to some remaining person.

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 1: solve using max flow (merging all points with the same angle into a single node with capacity = #points)
    1. Time complexity with Ford-Fulkerson is $O(p^2)$ where p is the number of train lines adjacent to some remaining person.
    2. However p is hard to analyze. Turns out that $p \approx \text{max coordinate} = 1000$, so this approach is fast enough.

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 2: greedyish solution

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 2: greedyish solution
    1. First cut the cycle anywhere to get a path, solve path with simple greedy

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 2: greedyish solution
   1. First cut the cycle anywhere to get a path, solve path with simple greedy
   2. Make a second greedy pass to adjust assignment across the point where we cut the cycle.

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 2: greedyish solution
    1. First cut the cycle anywhere to get a path, solve path with simple greedy
    2. Make a second greedy pass to adjust assignment across the point where we cut the cycle.
    3. Time complexity is $O(n \log n)$.

H — Hubtown

Problem

Given a set of rays from in 2D $(0, 0)$, and some points, assign max #points to a ray of minimum angular distance from the point, subject to ray capacities.

Phase 2

1. For remaining points (having two closest rays) we get a bipartite matching problem between points and rays.
2. Graph is very simple (either a cycle, or a collection of paths)
3. Approach 2: greedyish solution
   1. First cut the cycle anywhere to get a path, solve path with simple greedy
   2. Make a second greedy pass to adjust assignment across the point where we cut the cycle.
   3. Time complexity is $O(n \log n)$.

Statistics: 17 submissions, 0 accepted

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

1. A copy of F_{30} will have at least 2^{30} vertices (assuming F_0 has at least 2 leaves)

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

1. A copy of F_{30} will have at least 2^{30} vertices (assuming F_0 has at least 2 leaves)
2. If $k > 30$, only relevant part is bottom-left copy of F_{30} and the path to this subtree.

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

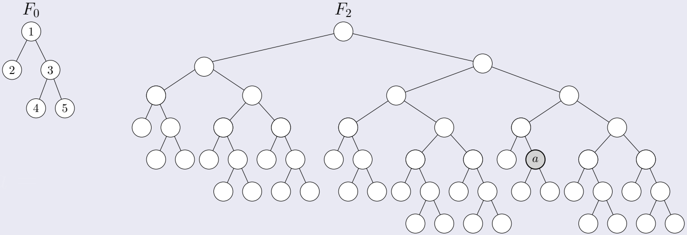
1. A copy of F_{30} will have at least 2^{30} vertices (assuming F_0 has at least 2 leaves)
2. If $k > 30$, only relevant part is bottom-left copy of F_{30} and the path to this subtree.
3. Reduces the problem to $k \leq 30$.

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution



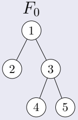
1. Representation of vertex a in the big tree:

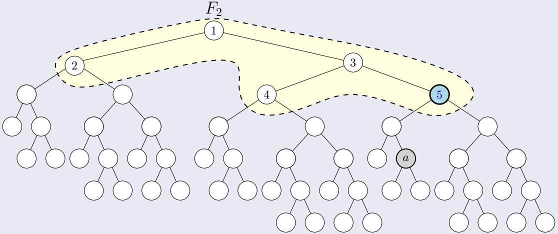
F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution





$a = (5,$

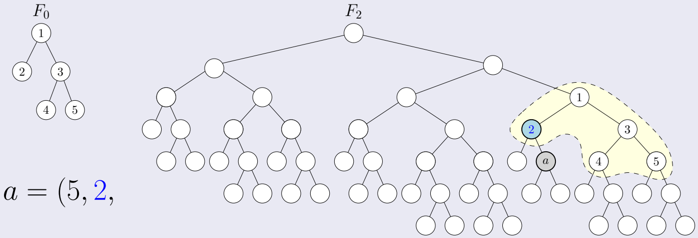
1. Representation of vertex a in the big tree:
    1. Record sequence $(a_1, a_2, \ldots)$ of leaves picked in each copy of F_0 when going from root to a

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution



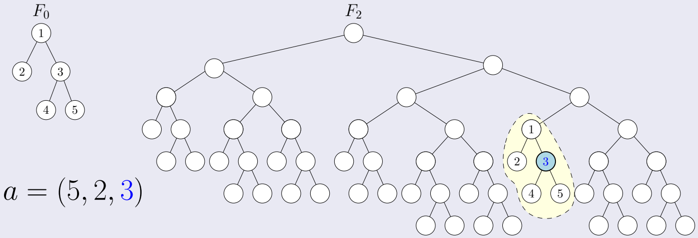
1. Representation of vertex a in the big tree:
   1. Record sequence $(a_1, a_2, \ldots)$ of leaves picked in each copy of F_0 when going from root to a

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution



1. Representation of vertex a in the big tree:
    1. Record sequence $(a_1, a_2, \ldots)$ of leaves picked in each copy of F_0 when going from root to a
    2. Finally add which node a corresponds to in last copy of F_0.

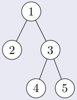
F — Fractal Tree

Problem

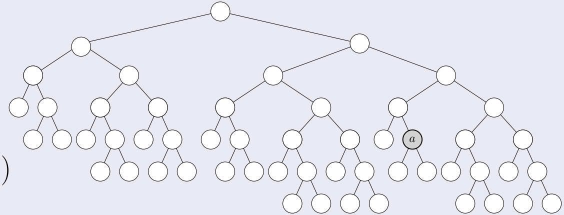
Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

F_0



F_2



$$a = (5, 2, 3)$$

1. Representation of vertex *a* in the big tree.
2. Can find this representation in $O(k \log n)$ time using binary search and precomputation of subtree sizes.

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

1. What is distance between $(a_1, \ldots, a_p)$ and $(b_1, \ldots, b_q)$?

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

1. What is distance between $(a_1, \ldots, a_p)$ and $(b_1, \ldots, b_q)$?
2. First remove any common prefix
   (moving into the same subtree does not affect distance)

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

1. What is distance between $(a_1, \ldots, a_p)$ and $(b_1, \ldots, b_q)$?
2. First remove any common prefix
   (moving into the same subtree does not affect distance)
3. Then, when $a_1 \neq b_1$, distance is
$$d(a_1, b_1) + \sum_{i=2}^{p} h(a_i) + \sum_{i=2}^{q} h(b_i)$$
4. $d(a_1, b_1) =$ distance between a_1 and b_1 in F_0
   (can compute it using a lowest common ancestor (LCA) algorithm)
5. $h(x) =$ depth of node x in F_0

F — Fractal Tree

Problem

Given a huge tree with potentially $100000^{2^{30}}$ vertices, find distances between pairs of vertices.

Solution

1. What is distance between $(a_1, \ldots, a_p)$ and $(b_1, \ldots, b_q)$?
2. First remove any common prefix
   (moving into the same subtree does not affect distance)
3. Then, when $a_1 \neq b_1$, distance is
$$d(a_1, b_1) + \sum_{i=2}^{p} h(a_i) + \sum_{i=2}^{q} h(b_i)$$
4. $d(a_1, b_1) =$ distance between a_1 and b_1 in F_0
   (can compute it using a lowest common ancestor (LCA) algorithm)
5. $h(x) =$ depth of node x in F_0

Statistics: 5 submissions, 0 accepted

Random numbers

253 teams

611 contestants

2819 total number of submissions

10 programming languages used by teams

Ordered by #submissions: C++ (1016), Java (865), Python (763), C (67), C# (65), Haskell (16), Prolog (9), Scala (8), Go (6), Ruby (4)

438 number of lines of code used in total by the shortest **jury** solutions to solve the entire problem set. (Significantly smaller than previous years – no killer problem in terms of implementation this year.)

Random facts

- All but two of the problems have **near-linear solutions**

  Exceptions:

Random facts

- All but two of the problems have **near-linear solutions**

  Exceptions:
  D (**Distinctive Character**) – $O(n + k \cdot 2^k)$ solution
  I (**Import Spaghetti**) – $O(n^3)$ solution.

Random facts

- All but two of the problems have **near-linear solutions**

  Exceptions:
  D (**Distinctive Character**) – $O(n + k \cdot 2^k)$ solution
  I (**Import Spaghetti**) – $O(n^3)$ solution.

- Two weeks ago, 3 people had written solutions to H (**Hubtown**). A day later, it had turned out that the 3 solutions were all wrong, with 3 completely different bugs, and that the test case generator had also been buggy.

Random facts

- All but two of the problems have **near-linear solutions**
  Exceptions:
  D (**Distinctive Character**) – $O(n + k \cdot 2^k)$ solution
  I (**Import Spaghetti**) – $O(n^3)$ solution.
- Two weeks ago, 3 people had written solutions to H (**Hubtown**). A day later, it had turned out that the 3 solutions were all wrong, with 3 completely different bugs, and that the test case generator had also been buggy.
  Two days ago, one of those Hubtown solutions again turned out to be wrong, more data was added.

Random facts

- All but two of the problems have **near-linear solutions**

  Exceptions:
  D (**Distinctive Character**) – $O(n + k \cdot 2^k)$ solution
  I (**Import Spaghetti**) – $O(n^3)$ solution.

- Two weeks ago, 3 people had written solutions to H (**Hubtown**). A day later, it had turned out that the 3 solutions were all wrong, with 3 completely different bugs, and that the test case generator had also been buggy.
  Two days ago, one of those Hubtown solutions again turned out to be wrong, more data was added.

- The jury wrote Python solutions for all problems except C (**Compass Card Sales**). But mostly in Python 2, which is faster than Python 3 on Kattis due to using pypy instead of CPython. The Python solutions are always the shortest (often by a wide margin).

What now?

- Northwestern Europe Regional Contest (NWERC):
  November 26 in Bath (UK).
  Teams from Nordic, Benelux, Germany, UK, Ireland.

- Each university sends up to two teams to NWERC to fight for spot in World Finals (April 2018, in Beijing, China)