DAPC 2023

Solutions presentation

The BAPC 2023 jury

September 23, 2023

F: Finding Forks

Problem Author: Ragnar Groot Koerkamp

Problem: Find the minimum number of forks that must have been in the dishwasher to get at least two empty places in the cutlery drawer.

F: Finding Forks

Problem Author: Ragnar Groot Koerkamp

Problem: Find the minimum number of forks that must have been in the dishwasher to get at least two empty places in the cutlery drawer.

Solution: Find the lowest two values in the input, and add them together.

F: Finding Forks

Problem Author: Ragnar Groot Koerkamp

Problem: Find the minimum number of forks that must have been in the dishwasher to get at least two empty places in the cutlery drawer.

Solution: Find the lowest two values in the input, and add them together.

Note: Can even be solved using 32-bit `int`, because $2 \cdot 10^9 < 2^{31}$.

F: Finding Forks

Problem Author: Ragnar Groot Koerkamp

Problem: Find the minimum number of forks that must have been in the dishwasher to get at least two empty places in the cutlery drawer.

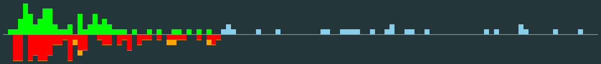
Solution: Find the lowest two values in the input, and add them together.

Note: Can even be solved using 32-bit `int`, because $2 \cdot 10^9 < 2^{31}$.

Statistics: 95 submissions, 72 accepted, 13 unknown

B: Better Dice

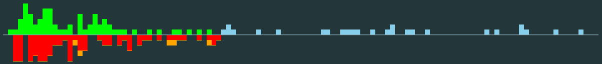
Problem Author: Mees de Vries



Problem: Determine which of the two custom die is more likely to roll a higher number.

B: Better Dice

Problem Author: Mees de Vries

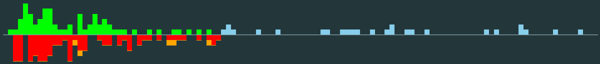


Problem: Determine which of the two custom die is more likely to roll a higher number.

Observation: The dice are fair and only have up to 1000 sides, so we can check all n^2 combinations.

B: Better Dice

Problem Author: Mees de Vries



Problem: Determine which of the two custom die is more likely to roll a higher number.

Observation: The dice are fair and only have up to 1000 sides, so we can check all n^2 combinations.

Solution: For every combination, count whether the first or second die is better.

Compare the total count for both dice to determine which die is more likely to roll a higher number.

B: Better Dice

Problem Author: Mees de Vries

Problem: Determine which of the two custom die is more likely to roll a higher number.

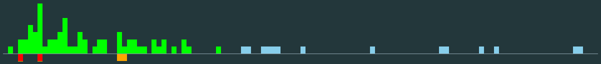
Observation: The dice are fair and only have up to 1000 sides, so we can check all n^2 combinations.

Solution: For every combination, count whether the first or second die is better.
Compare the total count for both dice to determine which die is more likely to roll a higher number.

Statistics: 172 submissions, 65 accepted, 26 unknown

J: Just a Joystick

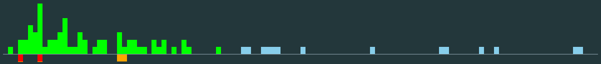
Problem Author: Maarten Sijm



Problem: How many times do you need to move the joystick up or down to enter your initials?

J: Just a Joystick

Problem Author: Maarten Sijm

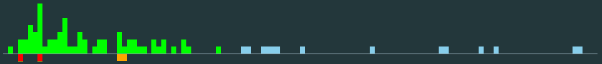


Problem: How many times do you need to move the joystick up or down to enter your initials?

Observation: Each letter position can be treated individually.

J: Just a Joystick

Problem Author: Maarten Sijm



Problem: How many times do you need to move the joystick up or down to enter your initials?

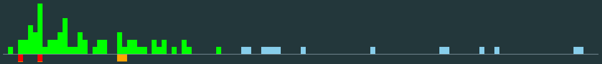
Observation: Each letter position can be treated individually.

Solution: Sum, for all pairs of letters, the “distance” on the alphabet wheel:

$$\sum_{(a,b)} \min(a-b \mod 26, b-a \mod 26)$$

J: Just a Joystick

Problem Author: Maarten Sijm



Problem: How many times do you need to move the joystick up or down to enter your initials?

Observation: Each letter position can be treated individually.

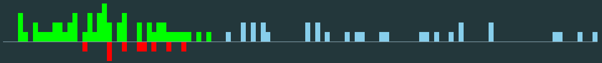
Solution: Sum, for all pairs of letters, the "distance" on the alphabet wheel:

$$\sum_{(a,b)} \min(a - b \mod 26, b - a \mod 26)$$

Statistics: 82 submissions, 64 accepted, 14 unknown

K: King's Keep

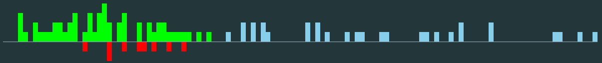
Problem Author: Ivan Fefer



Problem: Compute the minimal average distance from the most optimal residence keep to the other keeps.

K: King's Keep

Problem Author: Ivan Fefer

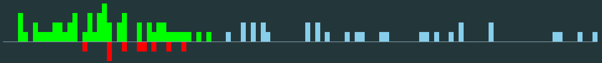


Problem: Compute the minimal average distance from the most optimal residence keep to the other keeps.

Observation: There are $k \leq 1000$ keeps, so $\mathcal{O}(k^2)$ is fine.

K: King's Keep

Problem Author: Ivan Fefer



Problem: Compute the minimal average distance from the most optimal residence keep to the other keeps.

Observation: There are $k \leq 1000$ keeps, so $\mathcal{O}(k^2)$ is fine.

Solution: For every keep, calculate the average distance to all other keeps, and take the minimum:

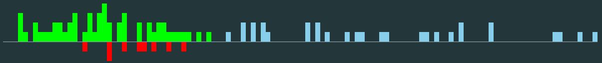
$$\min_{1 \leq i \leq k} \left(\frac{\sum_{j \neq i} d(i,j)}{k-1} \right)$$

$d(i,j)$ is Euclidean distance between keeps i and j:

$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

K: King's Keep

Problem Author: Ivan Fefer



Problem: Compute the minimal average distance from the most optimal residence keep to the other keeps.

Observation: There are $k \leq 1000$ keeps, so $\mathcal{O}(k^2)$ is fine.

Solution: For every keep, calculate the average distance to all other keeps, and take the minimum:

$$\min_{1 \leq i \leq k} \left(\frac{\sum_{j \neq i} d(i,j)}{k-1} \right)$$

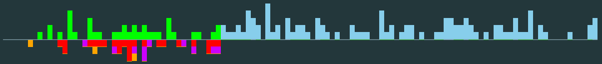
$d(i,j)$ is Euclidean distance between keeps i and j:

$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Statistics: 93 submissions, 54 accepted, 30 unknown

I: Idle Terminal

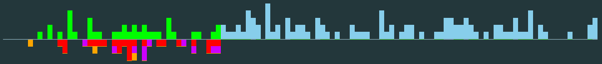
Problem Author: Ragnar Groot Koerkamp



Problem: Calculate the longest time that goes by without seeing a new message on the terminal.

I: Idle Terminal

Problem Author: Ragnar Groot Koerkamp



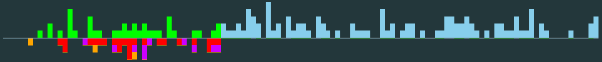
Problem: Calculate the longest time that goes by without seeing a new message on the terminal.

Solution: Simulate the processing of the migration jobs and find the largest gap.

- For each of the n jobs, find the first available CPU core, and update this core's end time.
- Make sure to correctly handle the start and end of the simulation.

I: Idle Terminal

Problem Author: Ragnar Groot Koerkamp



Problem: Calculate the longest time that goes by without seeing a new message on the terminal.

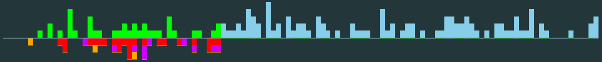
Solution: Simulate the processing of the migration jobs and find the largest gap.

- For each of the n jobs, find the first available CPU core,
  and update this core's end time.
- Make sure to correctly handle the start and end of the simulation.

Pitfall: Finding the first available CPU core in a list ($\mathcal{O}(k)$ time) is too slow,
use a priority queue instead ($\mathcal{O}(\log k)$ time).

I: Idle Terminal

Problem Author: Ragnar Groot Koerkamp



Problem: Calculate the longest time that goes by without seeing a new message on the terminal.

Solution: Simulate the processing of the migration jobs and find the largest gap.

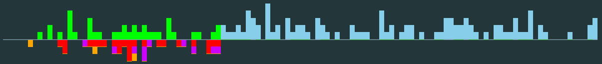
- For each of the n jobs, find the first available CPU core, and update this core's end time.
- Make sure to correctly handle the start and end of the simulation.

Pitfall: Finding the first available CPU core in a list ($\mathcal{O}(k)$ time) is too slow, use a priority queue instead ($\mathcal{O}(\log k)$ time).

Run time: $\mathcal{O}(n \log k)$

I: Idle Terminal

Problem Author: Ragnar Groot Koerkamp



Problem: Calculate the longest time that goes by without seeing a new message on the terminal.

Solution: Simulate the processing of the migration jobs and find the largest gap.

- For each of the n jobs, find the first available CPU core, and update this core's end time.
- Make sure to correctly handle the start and end of the simulation.

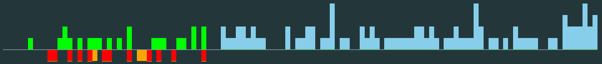
Pitfall: Finding the first available CPU core in a list ($\mathcal{O}(k)$ time) is too slow, use a priority queue instead ($\mathcal{O}(\log k)$ time).

Run time: $\mathcal{O}(n \log k)$

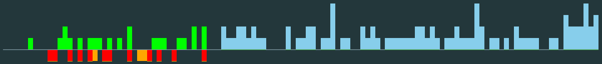
Statistics: 174 submissions, 35 accepted, 101 unknown

A: Anti-Tetris

Problem Author: Maarten Sijm



Problem: Design a *Tetris* grid that perfectly fits the input block.



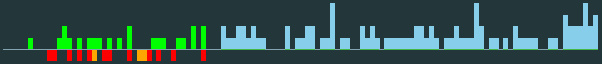
Problem: Design a *Tetris* grid that perfectly fits the input block.

Observation: The grid can only become perfect if the block has a side with only ‘**#**’.

- (such a side should be at the top)

A: Anti-Tetris

Problem Author: Maarten Sijm



Problem: Design a *Tetris* grid that perfectly fits the input block.

Observation: The grid can only become perfect if the block has a side with only ‘**#**’.

- (such a side should be at the top)

Solution: Find a side that has only ‘**#**’.

A: Anti-Tetris

Problem Author: Maarten Sijm

Problem: Design a *Tetris* grid that perfectly fits the input block.

Observation: The grid can only become perfect if the block has a side with only ‘**#**’.

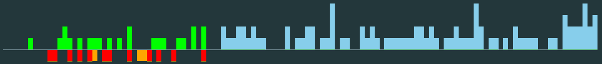
- (such a side should be at the top)

Solution: Find a side that has only ‘**#**’.

Rotate the block to have this side point upwards.

A: Anti-Tetris

Problem Author: Maarten Sijm



Problem: Design a *Tetris* grid that perfectly fits the input block.

Observation: The grid can only become perfect if the block has a side with only ‘**#**’.

- (such a side should be at the top)

Solution: Find a side that has only ‘**#**’.

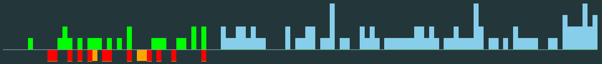
Rotate the block to have this side point upwards.

Verify that the block has no holes.

- Each column should have only ‘**#**’ at the top, followed by ‘**.**’ at the bottom.

A: Anti-Tetris

Problem Author: Maarten Sijm



Problem: Design a *Tetris* grid that perfectly fits the input block.

Observation: The grid can only become perfect if the block has a side with only '**#**'.

- (such a side should be at the top)

Solution: Find a side that has only '**#**'.

Rotate the block to have this side point upwards.

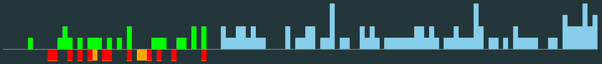
Verify that the block has no holes.

- Each column should have only '**#**' at the top, followed by '**.**' at the bottom.

Invert the block (i.e. swap '**#**' and '**.**') to get a grid that it would fit in.

A: Anti-Tetris

Problem Author: Maarten Sijm



Problem: Design a *Tetris* grid that perfectly fits the input block.

Observation: The grid can only become perfect if the block has a side with only ‘**#**’.

- (such a side should be at the top)

Solution: Find a side that has only ‘**#**’.

Rotate the block to have this side point upwards.

Verify that the block has no holes.

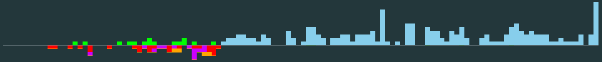
- Each column should have only ‘**#**’ at the top, followed by ‘**.**’ at the bottom.

Invert the block (i.e. swap ‘**#**’ and ‘**.**’) to get a grid that it would fit in.

Statistics: 129 submissions, 22 accepted, 91 unknown

C: Cheap Flying

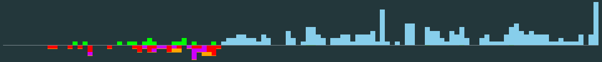
Problem Author: Gregor Behnke



Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

C: Cheap Flying

Problem Author: Gregor Behnke

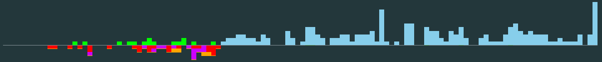


Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

Observation: After `buying` your own aircraft, always fly `yourself`.

C: Cheap Flying

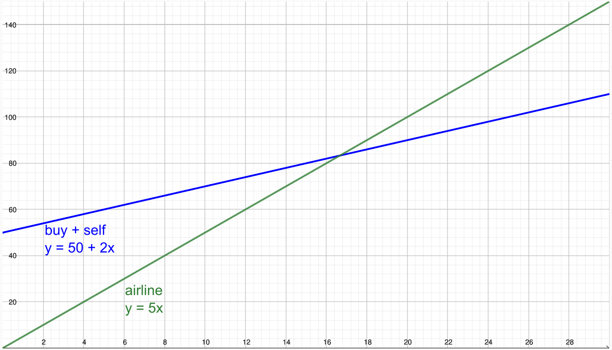
Problem Author: Gregor Behnke



Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly your`self`, keeping the cost below twice the optimum.

Observation: After `buy`ing your own aircraft, always fly your`self`.

To solve: When to `buy` your aircraft?

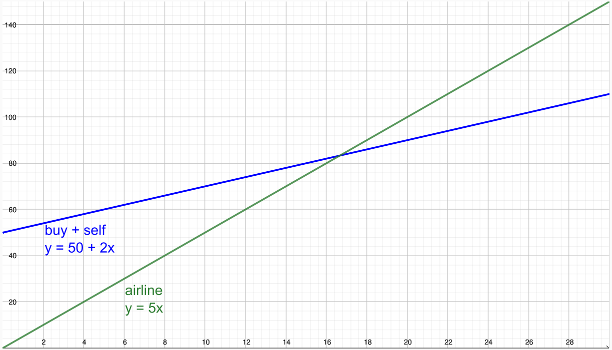


Source: Geogebra.org

Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

Observation: After `buy`ing your own aircraft, always fly `yourself`.

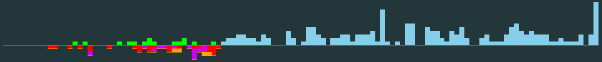
To solve: When to `buy` your aircraft? $\Rightarrow$ "`buy`" when $b + cx < ax$.



Source: Geogebra.org

C: Cheap Flying

Problem Author: Gregor Behnke



Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

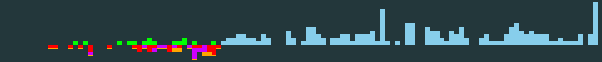
Observation: After `buying` your own aircraft, always fly `yourself`.

To solve: When to `buy` your aircraft? $\Rightarrow$ "`buy`" when $b + cx < ax$.

Solution: Print "`airline`" until the cost becomes higher than flying yourself. Then, print "`buy`", followed by printing "`self`" until you read "`end`".

C: Cheap Flying

Problem Author: Gregor Behnke



Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

Observation: After `buy`ing your own aircraft, always fly `yourself`.

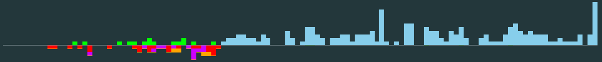
To solve: When to `buy` your aircraft? $\Rightarrow$ "`buy`" when $b + cx < ax$.

Solution: Print "`airline`" until the cost becomes higher than flying yourself. Then, print "`buy`", followed by printing "`self`" until you read "`end`".

Edge cases: $0 \leq a, b, c \leq 10^6$, so for example, it is possible that $a > b + c$ (immediately "`buy`") or $a = b = c = 0$ (always "`airline`").

C: Cheap Flying

Problem Author: Gregor Behnke



Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

Observation: After `buy`ing your own aircraft, always fly `yourself`.

To solve: When to `buy` your aircraft? $\Rightarrow$ "`buy`" when $b + cx < ax$.

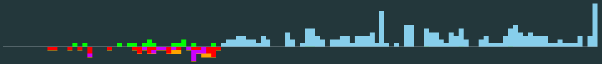
Solution: Print "`airline`" until the cost becomes higher than flying yourself. Then, print "`buy`", followed by printing "`self`" until you read "`end`".

Edge cases: $0 \leq a, b, c \leq 10^6$, so for example, it is possible that $a > b + c$ (immediately "`buy`") or $a = b = c = 0$ (always "`airline`").

Note: We did not specify the exact number of interactions up front. So, we check the cost condition after every `flight`.

C: Cheap Flying

Problem Author: Gregor Behnke



Problem: On the fly, decide whether to use the `airline` or `buy` your own aircraft and fly `yourself`, keeping the cost below twice the optimum.

Observation: After `buy`ing your own aircraft, always fly `yourself`.

To solve: When to `buy` your aircraft? $\Rightarrow$ "`buy`" when $b + cx < ax$.

Solution: Print "`airline`" until the cost becomes higher than flying yourself.
Then, print "`buy`", followed by printing "`self`" until you read "`end`".

Edge cases: $0 \leq a, b, c \leq 10^6$, so for example, it is possible that $a > b + c$ (immediately "`buy`") or $a = b = c = 0$ (always "`airline`").

Note: We did not specify the exact number of interactions up front.
So, we check the cost condition after every `flight`.

Statistics: 245 submissions, 15 accepted, 189 unknown

H: Hacky Ordering

Problem Author: Jorke de Vlas

Problem: Find a permutation of the English alphabet such that the strings are sorted.

H: Hacky Ordering

Problem Author: Jorke de Vlas

Problem: Find a permutation of the English alphabet such that the strings are sorted.

Observation: Trying all permutations is too slow, but many permutations will be killed early.

H: Hacky Ordering

Problem Author: Jorke de Vlas

Problem: Find a permutation of the English alphabet such that the strings are sorted.

Observation: Trying all permutations is too slow, but many permutations will be killed early.

Preparation: Instead of running over the entire list of words every time, create a graph, adding edges between the first differing pair of letters of two adjacent words:

```
cplusplus
csharp
python
php
```

c   y

h

$p \longrightarrow s$

H: Hacky Ordering

Problem Author: Jorke de Vlas

Problem: Find a permutation of the English alphabet such that the strings are sorted.

Observation: Trying all permutations is too slow, but many permutations will be killed early.

Preparation: Instead of running over the entire list of words every time, create a graph, adding edges between the first differing pair of letters of two adjacent words:

```
cplusplus
csharp
python
php
```

$$\begin{tikzcd} \mathtt{c} \arrow[d] & \mathtt{y} & \\ & & \mathtt{h} \\ \mathtt{p} \arrow[r] & \mathtt{s} & \end{tikzcd}$$

H: Hacky Ordering

Problem Author: Jorke de Vlas

Problem: Find a permutation of the English alphabet such that the strings are sorted.

Observation: Trying all permutations is too slow, but many permutations will be killed early.

Preparation: Instead of running over the entire list of words every time, create a graph, adding edges between the first differing pair of letters of two adjacent words:

```
cplusplus
csharp
python
php
```

$$\begin{tikzcd} c \arrow[dd] & y \arrow[dr] & \\ & & h \\ p \arrow[r] & s & \end{tikzcd}$$

Problem: Find a permutation of the English alphabet such that the strings are sorted.

Observation: Trying all permutations is too slow, but many permutations will be killed early.

Preparation: Instead of running over the entire list of words every time, create a graph, adding edges between the first differing pair of letters of two adjacent words:

```
cplusplus
csharp
python
php
```

$$\begin{tikzcd}
\mathtt{c} \arrow[dd] & \mathtt{y} \arrow[dr] & \\
& & \mathtt{h} \\
\mathtt{p} \arrow[r] & \mathtt{s} &
\end{tikzcd}$$

Solution: If the graph contains a cycle, print "`impossible`".
Else, print the reverse order of a post-order traversal of the graph.

H: Hacky Ordering

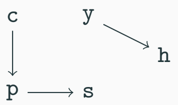
Problem Author: Jorke de Vlas

Problem: Find a permutation of the English alphabet such that the strings are sorted.

Observation: Trying all permutations is too slow, but many permutations will be killed early.

Preparation: Instead of running over the entire list of words every time, create a graph, adding edges between the first differing pair of letters of two adjacent words:

```
cplusplus
csharp
python
php
```

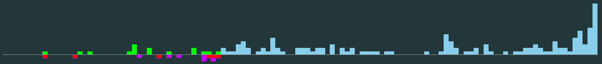


Solution: If the graph contains a cycle, print "`impossible`".
Else, print the reverse order of a post-order traversal of the graph.

Statistics: 117 submissions, 15 accepted, 89 unknown

E: Exceeding Limits

Problem Author: Maarten Sijm



Problem: Find the minimal amount of speeding to arrive on time.

E: Exceeding Limits

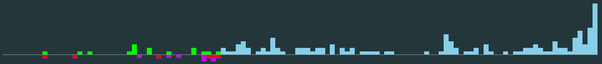
Problem Author: Maarten Sijm

Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

E: Exceeding Limits

Problem Author: Maarten Sijm



Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

Observation: Driving with different speeds may cause a different route to be faster.

E: Exceeding Limits

Problem Author: Maarten Sijm

Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

Observation: Driving with different speeds may cause a different route to be faster.

Observation: If you can reach the destination with some amount of speeding,
you can always reach the destination by speeding more.

E: Exceeding Limits

Problem Author: Maarten Sijm

Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

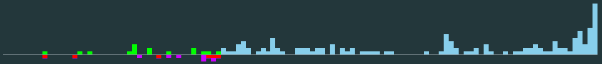
Observation: Driving with different speeds may cause a different route to be faster.

Observation: If you can reach the destination with some amount of speeding,
you can always reach the destination by speeding more.

Solution: Binary search on the amount of speeding, performing Dijkstra with the new speeds.
If destination can be reached on time, try higher; else, try lower.

E: Exceeding Limits

Problem Author: Maarten Sijm



Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

Observation: Driving with different speeds may cause a different route to be faster.

Observation: If you can reach the destination with some amount of speeding,
you can always reach the destination by speeding more.

Solution: Binary search on the amount of speeding, performing Dijkstra with the new speeds.
If destination can be reached on time, try higher; else, try lower.

Run time: $\mathcal{O}((m + n\log m) \cdot \log t)$.

E: Exceeding Limits

Problem Author: Maarten Sijm

Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

Observation: Driving with different speeds may cause a different route to be faster.

Observation: If you can reach the destination with some amount of speeding,
you can always reach the destination by speeding more.

Solution: Binary search on the amount of speeding, performing Dijkstra with the new speeds.
If destination can be reached on time, try higher; else, try lower.

Run time: $\mathcal{O}((m + n \log m) \cdot \log t)$.

Note: Floating-point precision is not a problem, because of the low bounds on t and v (10^5).

E: Exceeding Limits

Problem Author: Maarten Sijm

Problem: Find the minimal amount of speeding to arrive on time.

First check: Perform Dijkstra to check if the destination can be reached on time without speeding.

Observation: Driving with different speeds may cause a different route to be faster.

Observation: If you can reach the destination with some amount of speeding,
you can always reach the destination by speeding more.

Solution: Binary search on the amount of speeding, performing Dijkstra with the new speeds.
If destination can be reached on time, try higher; else, try lower.

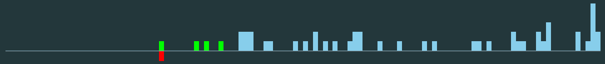
Run time: $\mathcal{O}((m + n \log m) \cdot \log t)$.

Note: Floating-point precision is not a problem, because of the low bounds on t and v (10^5).

Statistics: 160 submissions, 15 accepted, 133 unknown

G: Gathering Search Results

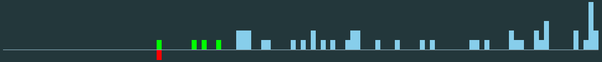
Problem Author: Pim Spelier



Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

G: Gathering Search Results

Problem Author: Pim Spelier



Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

Solution: Sort on average position. (Or equivalently: sum of positions)

Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

Solution: Sort on average position. (Or equivalently: sum of positions)

Proof:

- Denote the average position of result r by $\mu(r) = \frac{1}{k} \sum_{s=1}^{k} \sigma_s(r)$.

Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

Solution: Sort on average position. (Or equivalently: sum of positions)

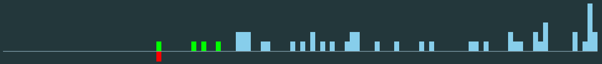
Proof:

- Denote the average position of result r by $\mu(r) = \frac{1}{k}\sum_{s=1}^{k} \sigma_s(r)$.
- A permutation τ has cost:

$$\begin{aligned}
\sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r)-\sigma_s(r))^2 &= \sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r)^2 - 2\tau(r)\sigma_s(r) + \sigma_s(r)^2) \\
&= \sum_{r=1}^{n}(k\tau(r)^2 - 2k\tau(r)\mu(r) + \text{constants}) \\
&= k\sum_{r=1}^{n}(\tau(r)-\mu(r))^2 + \text{other constant}
\end{aligned}$$

G: Gathering Search Results

Problem Author: Pim Spelier



Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

Solution: Sort on average position. (Or equivalently: sum of positions)

Proof:

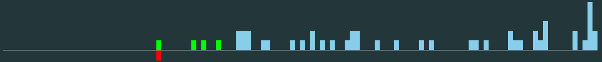
- Denote the average position of result r by $\mu(r) = \frac{1}{k}\sum_{s=1}^{k} \sigma_s(r)$.
- A permutation τ has cost:

$$\begin{aligned}
\sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r) - \sigma_s(r))^2 &= \sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r)^2 - 2\tau(r)\sigma_s(r) + \sigma_s(r)^2) \\
&= \sum_{r=1}^{n}(k\tau(r)^2 - 2k\tau(r)\mu(r) + \text{constants}) \\
&= k\sum_{r=1}^{n}(\tau(r) - \mu(r))^2 + \text{other constant}
\end{aligned}$$

- So $\sum_{r=1}^{n}(\tau(r) - \mu(r))^2$ needs to be minimized.

G: Gathering Search Results

Problem Author: Pim Spelier



Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

Solution: Sort on average position. (Or equivalently: sum of positions)

Proof:

- Denote the average position of result r by $\mu(r) = \frac{1}{k}\sum_{s=1}^{k} \sigma_s(r)$.
- A permutation τ has cost:

$$\begin{aligned}\sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r) - \sigma_s(r))^2 &= \sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r)^2 - 2\tau(r)\sigma_s(r) + \sigma_s(r)^2) \\ &= \sum_{r=1}^{n}(k\tau(r)^2 - 2k\tau(r)\mu(r) + \text{constants}) \\ &= k\sum_{r=1}^{n}(\tau(r) - \mu(r))^2 + \text{other constant}\end{aligned}$$

- So $\sum_{r=1}^{n}(\tau(r) - \mu(r))^2$ needs to be minimized.

Run time: $\mathcal{O}(nk + n\log n)$

G: Gathering Search Results

Problem Author: Pim Spelier

Problem: Given some permutations $\sigma_1, \ldots, \sigma_k$ of $\{1, \ldots, n\}$, determine a permutation such that the total cost is minimized.

Solution: Sort on average position. (Or equivalently: sum of positions)

Proof:

- Denote the average position of result r by $\mu(r) = \frac{1}{k}\sum_{s=1}^{k} \sigma_s(r)$.
- A permutation τ has cost:

$$\begin{aligned}\sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r) - \sigma_s(r))^2 &= \sum_{r=1}^{n}\sum_{s=1}^{k}(\tau(r)^2 - 2\tau(r)\sigma_s(r) + \sigma_s(r)^2)\\ &= \sum_{r=1}^{n}(k\tau(r)^2 - 2k\tau(r)\mu(r) + \text{constants})\\ &= k\sum_{r=1}^{n}(\tau(r) - \mu(r))^2 + \text{other constant}\end{aligned}$$

- So $\sum_{r=1}^{n}(\tau(r) - \mu(r))^2$ needs to be minimized.

Run time: $\mathcal{O}(nk + n\log n)$

Statistics: 51 submissions, 4 accepted, 46 unknown

D: Determining Duos

Problem Author: Pim Spelier

Problem: Given are the scores $x_{t,s}$ of $2n$ students on r topics, where for each topic the scores are a permutation of $\{1, \ldots, 2n\}$.

A pair (team) of students s_1, s_2 has team-score $S(s_1, s_2) := \sum_t \max(x_{t,s_1}, x_{t,s_2})$.

Is it possible to make pairs with total score $\frac{1}{2}rn(3n+1)$.

D: Determining Duos

Problem Author: Pim Spelier

Problem: Given are the scores $x_{t,s}$ of $2n$ students on r topics, where for each topic the scores are a permutation of $\{1, \ldots, 2n\}$.
A pair (team) of students s_1, s_2 has team-score $S(s_1, s_2) := \sum_t \max(x_{t,s_1}, x_{t,s_2})$.
Is it possible to make pairs with total score $\frac{1}{2}rn(3n+1)$.

Naive: This is general max-weighted matching in a complete graph on $2n$ vertices, where edge s_is_j has weight $S(s_i, s_j)$. (Complicated and too slow.)

D: Determining Duos

Problem Author: Pim Spelier

Problem: Given are the scores $x_{t,s}$ of $2n$ students on r topics, where for each topic the scores are a permutation of $\{1, \dots, 2n\}$.
A pair (team) of students s_1, s_2 has team-score $S(s_1, s_2) := \sum_t \max(x_{t,s_1}, x_{t,s_2})$.
Is it possible to make pairs with total score $\frac{1}{2}rn(3n+1)$.

Naive: This is general max-weighted matching in a complete graph on $2n$ vertices, where edge $s_i s_j$ has weight $S(s_i, s_j)$. (Complicated and too slow.)

Insight: What is the maximum possible total score per topic? I.e. for a permutation a of $\{1, \dots, 2n\}$, what is the maximum of

$$A = \max(a_1, a_2) + \max(a_3, a_4) + \cdots + \max(a_{2n-1}, a_{2n})?$$

D: Determining Duos

Problem Author: Pim Spelier

Problem: Given are the scores $x_{t,s}$ of $2n$ students on r topics, where for each topic the scores are a permutation of $\{1, \dots, 2n\}$.
A pair (team) of students s_1, s_2 has team-score $S(s_1, s_2) := \sum_t \max(x_{t,s_1}, x_{t,s_2})$.
Is it possible to make pairs with total score $\frac{1}{2}rn(3n+1)$.

Naive: This is general max-weighted matching in a complete graph on $2n$ vertices, where edge $s_i s_j$ has weight $S(s_i, s_j)$. (Complicated and too slow.)

Insight: What is the maximum possible total score per topic? I.e. for a permutation a of $\{1, \dots, 2n\}$, what is the maximum of

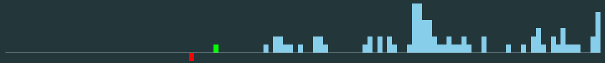
$$A = \max(a_1, a_2) + \max(a_3, a_4) + \cdots + \max(a_{2n-1}, a_{2n})?$$

Swap values such that $a_1 \leq a_2$, $a_3 \leq a_4$, ... Then $A = a_2 + a_4 + \cdots + a_{2n}$, which is maximal when

$$A \leq (n+1) + (n+2) + \cdots + (2n) = \frac{n \cdot ((n+1) + (2n))}{2} = \frac{1}{2}n(3n+1).$$

D: Determining Duos

Problem Author: Pim Spelier



Problem: Given are the scores $x_{t,s}$ of $2n$ students on r topics, where for each topic the scores are a permutation of $\{1, \dots, 2n\}$.
A pair (team) of students s_1, s_2 has team-score $S(s_1, s_2) := \sum_t \max(x_{t,s_1}, x_{t,s_2})$.
Is it possible to make pairs with total score $\frac{1}{2}rn(3n+1)$.

Naive: This is general max-weighted matching in a complete graph on $2n$ vertices, where edge $s_i s_j$ has weight $S(s_i, s_j)$. (Complicated and too slow.)

Insight: What is the maximum possible total score per topic? I.e. for a permutation a of $\{1, \dots, 2n\}$, what is the maximum of

$$A = \max(a_1, a_2) + \max(a_3, a_4) + \dots + \max(a_{2n-1}, a_{2n})?$$

Swap values such that $a_1 \le a_2$, $a_3 \le a_4$, ... Then $A = a_2 + a_4 + \dots + a_{2n}$, which is maximal when

$$A \le (n+1) + (n+2) + \dots + (2n) = \frac{n \cdot ((n+1) + (2n))}{2} = \frac{1}{2}n(3n+1).$$

Thus, $\frac{1}{2}rn(3n+1)$ is exactly the maximal possible score.

D: Determining Duos

Problem Author: Pim Spelier

Insight: The maximal pairing is only reached when for each pair of students (s_i, s_j) and each topic t, one of the scores x_{t,s_i} and x_{t,s_j} is *low* ($\leq n$) and the other is *high* ($> n$).

D: Determining Duos

Problem Author: Pim Spelier

Insight: The maximal pairing is only reached when for each pair of students (s_i, s_j) and each topic t, one of the scores x_{t,s_i} and x_{t,s_j} is *low* ($\leq n$) and the other is *high* ($> n$).

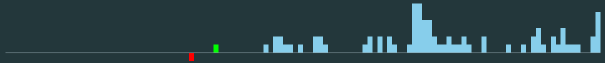
Solution: First convert the input to binary matrix indicating whether each score is low or high.

$$\begin{matrix} 1 & 4 & 2 & 5 & 2 & 6 \\ 1 & 4 & 5 & 6 & 2 & 3 \\ 1 & 4 & 5 & 6 & 2 & 3 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{matrix}$$

Now we must find a matching between *complementary* columns.

D: Determining Duos

Problem Author: Pim Spelier



Insight: The maximal pairing is only reached when for each pair of students (s_i, s_j) and each topic t, one of the scores x_{t,s_i} and x_{t,s_j} is *low* ($\leq n$) and the other is *high* ($> n$).

Solution: First convert the input to binary matrix indicating whether each score is low or high.

$$\begin{matrix} 1 & 4 & 2 & 5 & 2 & 6 \\ 1 & 4 & 5 & 6 & 2 & 3 \\ 1 & 4 & 5 & 6 & 2 & 3 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{matrix}$$

Now we must find a matching between *complementary* columns.

A matching exists iff each type of column has the same count as its complement.

D: Determining Duos

Problem Author: Pim Spelier

Insight: The maximal pairing is only reached when for each pair of students (s_i, s_j) and each topic t, one of the scores x_{t,s_i} and x_{t,s_j} is *low* $(\leq n)$ and the other is *high* $(> n)$.

Solution: First convert the input to binary matrix indicating whether each score is low or high.

$$\begin{matrix} 1 & 4 & 2 & 5 & 2 & 6 \\ 1 & 4 & 5 & 6 & 2 & 3 \\ 1 & 4 & 5 & 6 & 2 & 3 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{matrix}$$

Now we must find a matching between *complementary* columns.

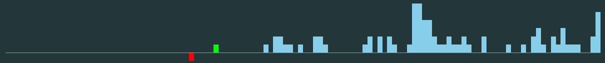
A matching exists iff each type of column has the same count as its complement.

Cute trick: Sort the columns, take the complement, and check if this equals the reverse.

$$\begin{matrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{matrix} \quad \longleftrightarrow \quad \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{matrix}$$

D: Determining Duos

Problem Author: Pim Spelier



Insight: The maximal pairing is only reached when for each pair of students (s_i, s_j) and each topic t, one of the scores x_{t,s_i} and x_{t,s_j} is *low* ($\leq n$) and the other is *high* ($> n$).

Solution: First convert the input to binary matrix indicating whether each score is low or high.

$$\begin{matrix} 1 & 4 & 2 & 5 & 2 & 6 \\ 1 & 4 & 5 & 6 & 2 & 3 \\ 1 & 4 & 5 & 6 & 2 & 3 \end{matrix} \longrightarrow \begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{matrix}$$

Now we must find a matching between *complementary* columns.

A matching exists iff each type of column has the same count as its complement.

Cute trick: Sort the columns, take the complement, and check if this equals the reverse.

$$\begin{matrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{matrix} \longleftrightarrow \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{matrix}$$

Run time: $\mathcal{O}(nr\log(n))$.

D: Determining Duos

Problem Author: Pim Spelier

Insight: The maximal pairing is only reached when for each pair of students (s_i, s_j) and each topic t, one of the scores x_{t,s_i} and x_{t,s_j} is *low* $(\leq n)$ and the other is *high* $(> n)$.

Solution: First convert the input to binary matrix indicating whether each score is low or high.

$$\begin{matrix} 1 & 4 & 2 & 5 & 2 & 6 \\ 1 & 4 & 5 & 6 & 2 & 3 \\ 1 & 4 & 5 & 6 & 2 & 3 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{matrix}$$

Now we must find a matching between *complementary* columns.

A matching exists iff each type of column has the same count as its complement.

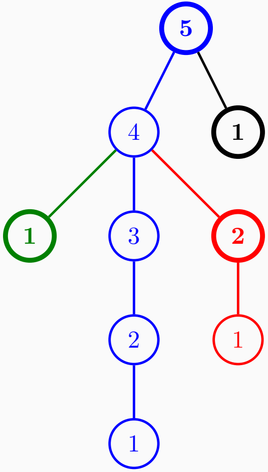
Cute trick: Sort the columns, take the complement, and check if this equals the reverse.

$$\begin{matrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{matrix} \quad \longleftrightarrow \quad \begin{matrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{matrix}$$

Run time: $\mathcal{O}(nr\log(n))$.

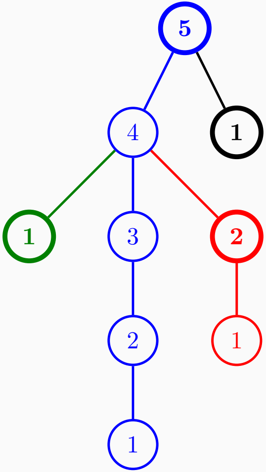
Statistics: 81 submissions, 1 accepted, 79 unknown

Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.



Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

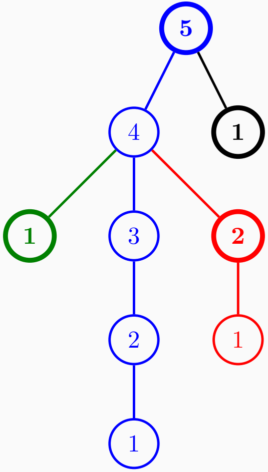
Insight: Removing a leaf only reduces the count if it has siblings.

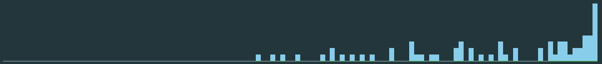


Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

Insight: Removing a leaf only reduces the count if it has siblings.

Greedy: Repeatedly remove the shortest *leaf-branch*.



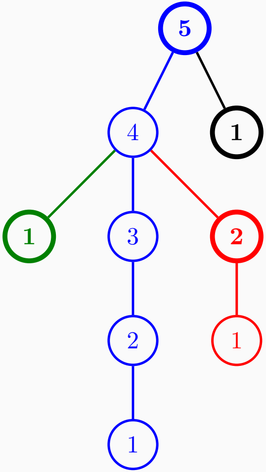


Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

Insight: Removing a leaf only reduces the count if it has siblings.

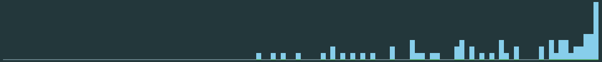
Greedy: Repeatedly remove the shortest *leaf-branch*.

Insight: Below each vertex, the deepest path is always removed last.



L: Losing Leaves

Problem Author: Ragnar Groot Koerkamp



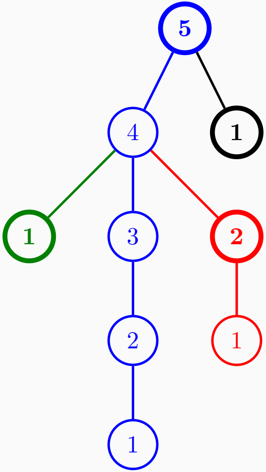
Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

Insight: Removing a leaf only reduces the count if it has siblings.

Greedy: Repeatedly remove the shortest *leaf-branch*.

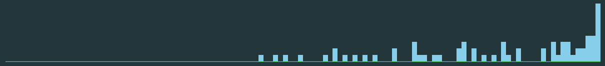
Insight: Below each vertex, the deepest path is always removed last.

Solution: Using DFS or bottom-up DP, find the length of the deepest path below each node.



L: Losing Leaves

Problem Author: Ragnar Groot Koerkamp



Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

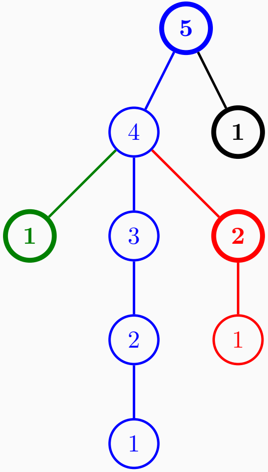
Insight: Removing a leaf only reduces the count if it has siblings.

Greedy: Repeatedly remove the shortest *leaf-branch*.

Insight: Below each vertex, the deepest path is always removed last.

Solution: Using DFS or bottom-up DP, find the length of the deepest path below each node.

At each node, increase the length of the deepest child by one, and mark the other children's paths as final (bold).



L: Losing Leaves

Problem Author: Ragnar Groot Koerkamp

Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

Insight: Removing a leaf only reduces the count if it has siblings.

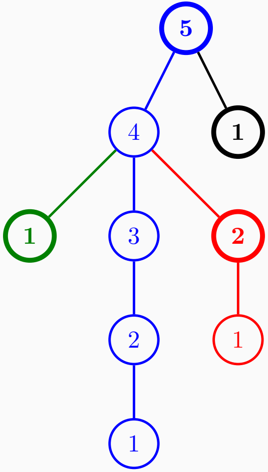
Greedy: Repeatedly remove the shortest *leaf-branch*.

Insight: Below each vertex, the deepest path is always removed last.

Solution: Using DFS or bottom-up DP, find the length of the deepest path below each node.

At each node, increase the length of the deepest child by one, and mark the other children's paths as final (bold).

Sort the final lengths ($[1, 1, 2, 5]$), and count how many of them sum to at most k.



L: Losing Leaves

Problem Author: Ragnar Groot Koerkamp

Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

Insight: Removing a leaf only reduces the count if it has siblings.

Greedy: Repeatedly remove the shortest *leaf-branch*.

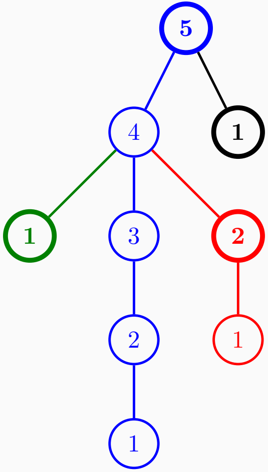
Insight: Below each vertex, the deepest path is always removed last.

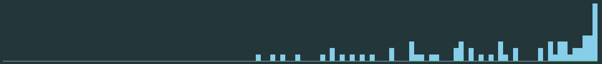
Solution: Using DFS or bottom-up DP, find the length of the deepest path below each node.

At each node, increase the length of the deepest child by one, and mark the other children's paths as final (bold).

Sort the final lengths ($[1, 1, 2, 5]$), and count how many of them sum to at most k.

Run time: $\mathcal{O}(n)$.





Problem: Given a tree of n vertices, remove k of them to minimize the number of remaining leaves.

Insight: Removing a leaf only reduces the count if it has siblings.

Greedy: Repeatedly remove the shortest *leaf-branch*.

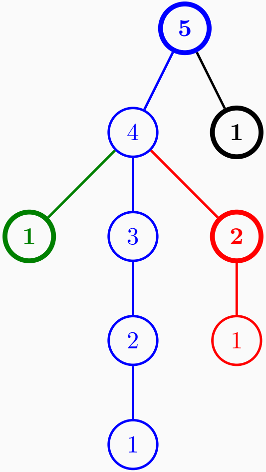
Insight: Below each vertex, the deepest path is always removed last.

Solution: Using DFS or bottom-up DP, find the length of the deepest path below each node.

At each node, increase the length of the deepest child by one, and mark the other children's paths as final (bold).

Sort the final lengths ($[1, 1, 2, 5]$), and count how many of them sum to at most k.

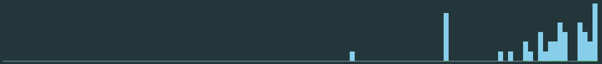
Run time: $\mathcal{O}(n)$.



Statistics: 69 submissions, 0 accepted, 69 unknown

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Problem: Given $n \leq 500$ trains that arrive at the north/south end of a one-lane tunnel, determine the minimal total waiting time.

M: Monorail

Problem Author: Ragnar Groot Koerkamp

Problem: Given $n \leq 500$ trains that arrive at the north/south end of a one-lane tunnel, determine the minimal total waiting time.

Insight: A train enters the tunnel either:

- *On time*: as soon as it arrives, or
- *Late*: directly after an opposite train exits the tunnel.

M: Monorail

Problem Author: Ragnar Groot Koerkamp

Problem: Given $n \leq 500$ trains that arrive at the north/south end of a one-lane tunnel, determine the minimal total waiting time.

Insight: A train enters the tunnel either:

- *On time*: as soon as it arrives, or
- *Late*: directly after an opposite train exits the tunnel.

Insight: After a train exits the tunnel, there are four possibilities for the next train:

1. Same direction and departs on time.
2. Opposite direction and enters at a later time (always on time).
3. Same direction and departs late, at the same time as current train.
4. Opposite direction and enters directly after (on time or late).

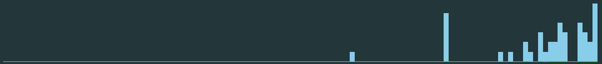
M: Monorail

Problem Author: Ragnar Groot Koerkamp

Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

M: Monorail

Problem Author: Ragnar Groot Koerkamp

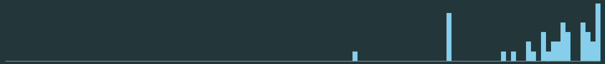


Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

Expand: Given state (N, i, j, T, W): i trains going north done; j trains going south done; last train went north and entered at time T; total waiting time W. Next possible states:

E1. $DP[N][i+1][j] \leq W$, when the next northbound train is on time $(N_{i+1} \geq T)$;

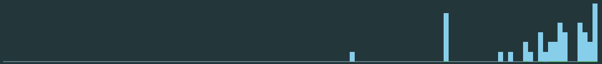
E2. $DP[S][i][j+1] \leq W$, when the next southbound train is on time $(S_{j+1} \geq T + D)$.

E3. $(N, i{+}1, j, T, W + (T - N_{i+1}))$, when next northbound train is late $(N_{i+1}{<}T)$;

E4. $(S, i, j{+}1, T{+}D, W + (N_i + D - S_{j+1}))$, when train $j{+}1$ leaves late $(S_{j+1}{<}T{+}D)$.

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

Expand: Given state (N, i, j, T, W): i trains going north done; j trains going south done; last train went north and entered at time T; total waiting time W. Next possible states:

E1. $DP[N][i+1][j] \leq W$, when the next northbound train is on time ($N_{i+1} \geq T$);

E2. $DP[S][i][j+1] \leq W$, when the next southbound train is on time ($S_{j+1} \geq T + D$).

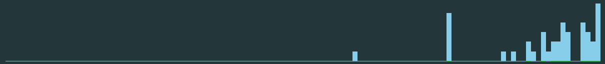
E3. $(N, i{+}1, j, T, W + (T - N_{i+1}))$, when next northbound train is late ($N_{i+1} < T$);

E4. $(S, i, j{+}1, T{+}D, W + (N_i + D - S_{j+1}))$, when train $j{+}1$ leaves late ($S_{j+1} < T{+}D$).

Greedy: When a train is late, send all other waiting trains as well. (Prefer E3 over E4.)

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

Expand: Given state (N, i, j, T, W): i trains going north done; j trains going south done; last train went north and entered at time T; total waiting time W. Next possible states:

E1. $DP[N][i+1][j] \leq W$, when the next northbound train is on time ($N_{i+1} \geq T$);

E2. $DP[S][i][j+1] \leq W$, when the next southbound train is on time ($S_{j+1} \geq T + D$).

E3. $(N, i{+}1, j, T, W + (T - N_{i+1}))$, when next northbound train is late ($N_{i+1} {<} T$);

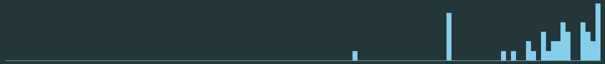
E4. $(S, i, j{+}1, T{+}D, W + (N_i + D - S_{j+1}))$, when train $j{+}1$ leaves late ($S_{j+1} {<} T{+}D$).

Greedy: When a train is late, send all other waiting trains as well. (Prefer E3 over E4.)

Recursion: For each DP state, consider all $\leq n$ states reached from it by alternating late trains from both sides (E3 and E4) and update DP via E1 and E2.
(See jury submissions for details.)

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

Expand: Given state (N, i, j, T, W): i trains going north done; j trains going south done; last train went north and entered at time T; total waiting time W. Next possible states:

- E1. $DP[N][i+1][j] \leq W$, when the next northbound train is on time $(N_{i+1} \geq T)$;
- E2. $DP[S][i][j+1] \leq W$, when the next southbound train is on time $(S_{j+1} \geq T + D)$.
- E3. $(N, i{+}1, j, T, W + (T - N_{i+1}))$, when next northbound train is late $(N_{i+1} {<} T)$;
- E4. $(S, i, j{+}1, T{+}D, W + (N_i + D - S_{j+1}))$, when train $j{+}1$ leaves late $(S_{j+1} {<} T{+}D)$.

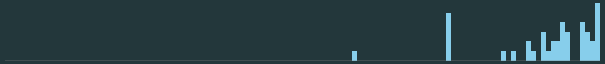
Greedy: When a train is late, send all other waiting trains as well. (Prefer E3 over E4.)

Recursion: For each DP state, consider all $\leq n$ states reached from it by alternating late trains from both sides (E3 and E4) and update DP via E1 and E2.
(See jury submissions for details.)

Run time: $\mathcal{O}(n^3)$: $\mathcal{O}(n^2)$ DP states with $\mathcal{O}(n)$ recursion in each.

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

Expand: Given state (N, i, j, T, W): i trains going north done; j trains going south done; last train went north and entered at time T; total waiting time W. Next possible states:

E1. $DP[N][i+1][j] \leq W$, when the next northbound train is on time $(N_{i+1} \geq T)$;

E2. $DP[S][i][j+1] \leq W$, when the next southbound train is on time $(S_{j+1} \geq T + D)$.

E3. $(N, i{+}1, j, T, W + (T - N_{i+1}))$, when next northbound train is late $(N_{i+1}{<}T)$;

E4. $(S, i, j{+}1, T{+}D, W + (N_i + D - S_{j+1}))$, when train $j{+}1$ leaves late $(S_{j+1}{<}T{+}D)$.

Greedy: When a train is late, send all other waiting trains as well. (Prefer E3 over E4.)

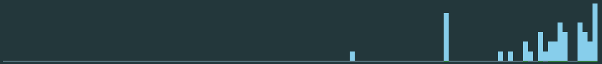
Recursion: For each DP state, consider all $\leq n$ states reached from it by alternating late trains from both sides (E3 and E4) and update DP via E1 and E2.
(See jury submissions for details.)

Run time: $\mathcal{O}(n^3)$: $\mathcal{O}(n^2)$ DP states with $\mathcal{O}(n)$ recursion in each.

Challenge: $\mathcal{O}(n^2)$ is also possible!

M: Monorail

Problem Author: Ragnar Groot Koerkamp



Solution: *Forward* DP: $DP[d][i][j]$ is the minimal total waiting time for the first i trains going north and j trains going south where the last train is in direction d and leaves on time.

Notation: N_i, S_j: arrival time of ith train north / jth train south. D: duration in tunnel.

Expand: Given state (N, i, j, T, W): i trains going north done; j trains going south done; last train went north and entered at time T; total waiting time W. Next possible states:

E1. $DP[N][i+1][j] \leq W$, when the next northbound train is on time $(N_{i+1} \geq T)$;

E2. $DP[S][i][j+1] \leq W$, when the next southbound train is on time $(S_{j+1} \geq T + D)$.

E3. $(N, i{+}1, j, T, W + (T - N_{i+1}))$, when next northbound train is late $(N_{i+1} {<} T)$;

E4. $(S, i, j{+}1, T{+}D, W + (N_i + D - S_{j+1}))$, when train $j{+}1$ leaves late $(S_{j+1} {<} T{+}D)$.

Greedy: When a train is late, send all other waiting trains as well. (Prefer E3 over E4.)

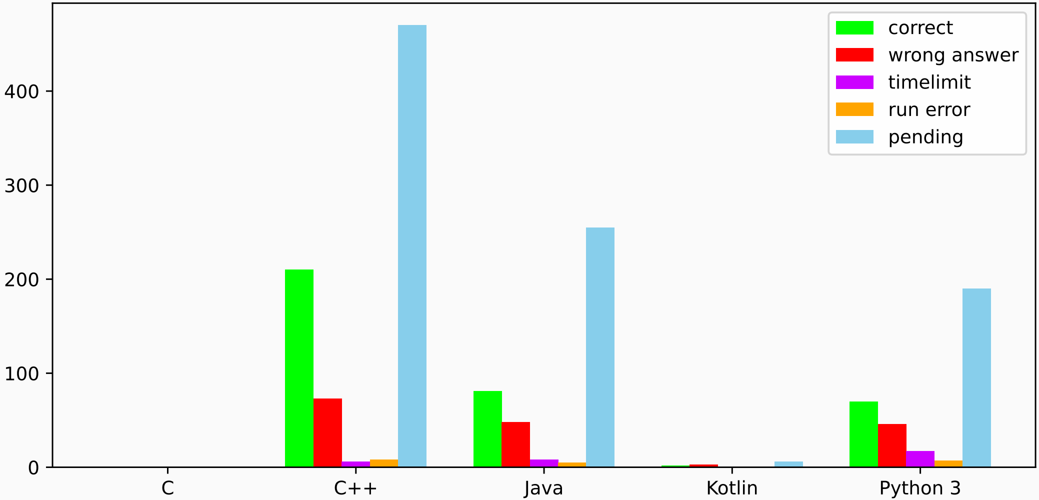
Recursion: For each DP state, consider all $\leq n$ states reached from it by alternating late trains from both sides (E3 and E4) and update DP via E1 and E2.
(See jury submissions for details.)

Run time: $\mathcal{O}(n^3)$: $\mathcal{O}(n^2)$ DP states with $\mathcal{O}(n)$ recursion in each.

Challenge: $\mathcal{O}(n^2)$ is also possible!

Statistics: 41 submissions, 0 accepted, 41 unknown

Language stats



Random facts

Jury work

- 492 commits (last year: 285)

[1] With limited codegolfing

Random facts

Jury work

- 492 commits (last year: 285)
- 1050 secret test cases (last year: 375) ($\approx$ 81 per problem!)

[1] With limited codegolfing

Random facts

Jury work

- 492 commits (last year: 285)
- 1050 secret test cases (last year: 375) ($\approx$ 81 per problem!)
- 195 jury + proofreader solutions (last year: 153)

[1] With limited codegolfing

Random facts

Jury work

- 492 commits (last year: 285)
- 1050 secret test cases (last year: 375) ($\approx$ 81 per problem!)
- 195 jury + proofreader solutions (last year: 153)
- The minimum[1] number of lines the jury needed to solve all problems is

$$5 + 14 + 15 + 4 + 21 + 2 + 10 + 30 + 9 + 2 + 3 + 18 + 48 = 181$$

  On average 13.9 lines per problem, up from 6.6 in last year's preliminaries

[1] With limited codegolfing

ETV also did their best!

Thanks to:

The proofreaders

Angel Karchev
Boas Kluiving
Jaap Eldering
Kevin Verbeek
Mark van Helvoort (Java Hero)
Michael Vasseur
Michael Zündorf
Nicky Gerritsen (Java Hero)
Paul Wild
Pavel Kuvnyavskiy (Kotlin Hero)
Thomas Verwoerd (Kotlin Hero)

The jury

Gregor Behnke
Ivan Fefer
Jorke de Vlas
Ludo Pulles
Maarten Sijm
Mees de Vries
Mike de Vries
Ragnar Groot Koerkamp
Reinier Schmiermann
Wessel van Woerden