Benelux Algorithm Programming Contest (BAPC) 2024

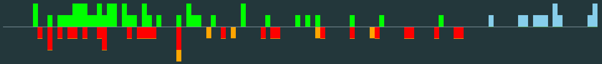
Solutions presentation

The BAPC 2024 jury

October 29, 2024

A: "Aaawww..." or "Aaayyy!!!"

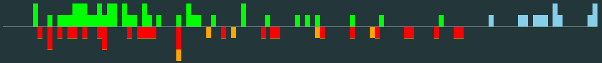
Problem author: Freek Henstra



Problem: Determine the final rank of your favourite team based on audience chants.

A: “Aaawww...” or “Aaayyy!!!”

Problem author: Freek Henstra



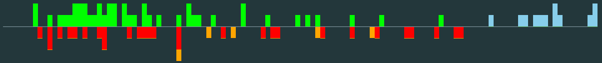
Problem: Determine the final rank of your favourite team based on audience chants.

Solution: Simulate the resolving of the scoreboard. For each audience chant:

- Look at the lowest ranking team with pending submissions.
- If the audience chant ends with exclamation marks, move the team up the list (number of ys $-$ 3) positions.

A: “Aaawww...” or “Aaayyy!!!”

Problem author: Freek Henstra



Problem: Determine the final rank of your favourite team based on audience chants.

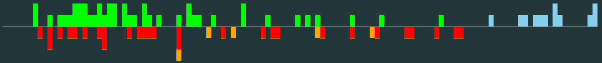
Solution: Simulate the resolving of the scoreboard. For each audience chant:

- Look at the lowest ranking team with pending submissions.
- If the audience chant ends with exclamation marks, move the team up the list (number of ys $- 3$) positions.

Simplification: For each team, you only need the number of pending submissions and their position, not the full state of the scoreboard.

A: “Aaawww...” or “Aaayyy!!!”

Problem author: Freek Henstra



Problem: Determine the final rank of your favourite team based on audience chants.

Solution: Simulate the resolving of the scoreboard. For each audience chant:

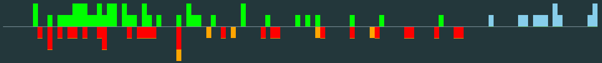
- Look at the lowest ranking team with pending submissions.
- If the audience chant ends with exclamation marks, move the team up the list (number of ys − 3) positions.

Simplification: For each team, you only need the number of pending submissions and their position, not the full state of the scoreboard.

Running time: $\mathcal{O}(n^2 m)$. But the limits on n and m are 100, so could even be as slow as $\mathcal{O}(n^2 m^2)$.

A: “Aaawww...” or “Aaayyy!!!”

Problem author: Freek Henstra



Problem: Determine the final rank of your favourite team based on audience chants.

Solution: Simulate the resolving of the scoreboard. For each audience chant:

- Look at the lowest ranking team with pending submissions.
- If the audience chant ends with exclamation marks, move the team up the list (number of `y`s $- 3$) positions.

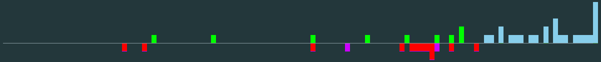
Simplification: For each team, you only need the number of pending submissions and their position, not the full state of the scoreboard.

Running time: $\mathcal{O}(n^2 m)$. But the limits on n and m are 100, so could even be as slow as $\mathcal{O}(n^2 m^2)$.

Statistics: 90 submissions, 44 accepted, 12 unknown

B: Buggy Blinkers

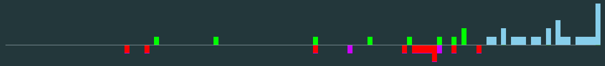
Problem author: Gijs Pennings



Problem: Find the shortest path in an unweighted graph, but you cannot make more than k uninterrupted turns.

B: Buggy Blinkers

Problem author: Gijs Pennings



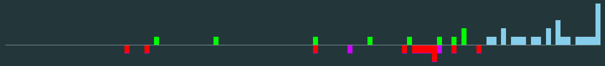
Problem: Find the shortest path in an unweighted graph, but you cannot make more than k uninterrupted turns.

Observations:

- An elementary BFS does not suffice, since you might exceed the turn limit.
- To determine when to activate the blinkers, you must keep track of the direction of arrival and current blinker state.

B: Buggy Blinkers

Problem author: Gijs Pennings



Problem: Find the shortest path in an unweighted graph, but you cannot make more than k uninterrupted turns.

Observations:

- An elementary BFS does not suffice, since you might exceed the turn limit.
- To determine when to activate the blinkers, you must keep track of the direction of arrival and current blinker state.

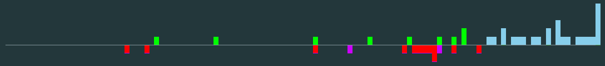
Solution: Perform a (breadth-first) search on a higher-dimensional space, where each "hypernode" is defined by

$$\langle \text{intersection}, \text{arrival direction}, \#\text{activations}, \text{blinker state} \rangle.$$

Prune the search if $\#\text{activations} > k$.

B: Buggy Blinkers

Problem author: Gijs Pennings



Problem: Find the shortest path in an unweighted graph, but you cannot make more than k uninterrupted turns.

Observations:

- An elementary BFS does not suffice, since you might exceed the turn limit.
- To determine when to activate the blinkers, you must keep track of the direction of arrival and current blinker state.

Solution: Perform a (breadth-first) search on a higher-dimensional space, where each "hypernode" is defined by

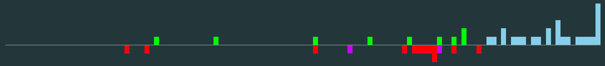
$$\langle \text{intersection}, \text{arrival direction}, \#\text{activations}, \text{blinker state} \rangle.$$

Prune the search if $\#\text{activations} > k$.

Running time: With $n \cdot 4 \cdot k \cdot 3$ hypernodes and each node having $\mathcal{O}(1)$ edges, BFS takes $\mathcal{O}(kn)$ time. Dijkstra with running time $\mathcal{O}(kn \log n)$ is accepted, but not necessary.

B: Buggy Blinkers

Problem author: Gijs Pennings



Problem: Find the shortest path in an unweighted graph, but you cannot make more than k uninterrupted turns.

Observations:

- An elementary BFS does not suffice, since you might exceed the turn limit.
- To determine when to activate the blinkers, you must keep track of the direction of arrival and current blinker state.

Solution: Perform a (breadth-first) search on a higher-dimensional space, where each "hypernode" is defined by

$$\langle \text{intersection}, \text{arrival direction}, \#\text{activations}, \text{blinker state} \rangle.$$

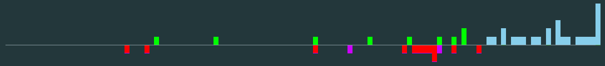
Prune the search if $\#\text{activations} > k$.

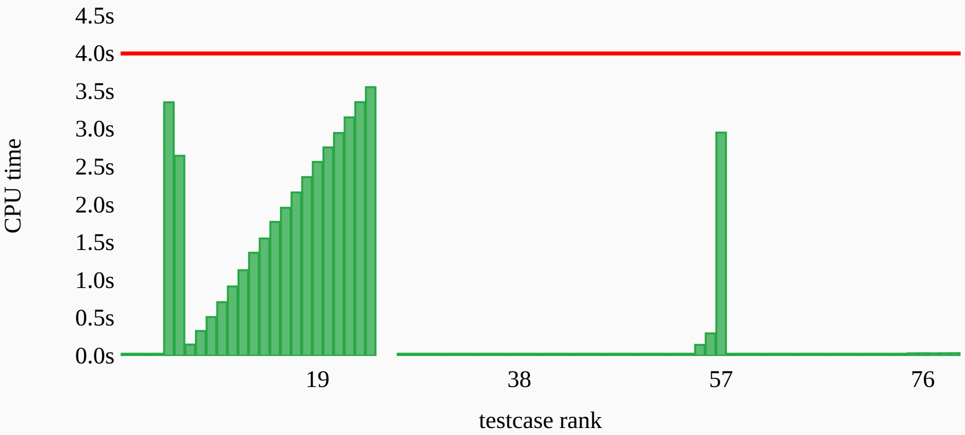
Running time: With $n \cdot 4 \cdot k \cdot 3$ hypernodes and each node having $\mathcal{O}(1)$ edges, BFS takes $\mathcal{O}(kn)$ time. Dijkstra with running time $\mathcal{O}(kn \log n)$ is accepted, but not necessary.

Statistics: 49 submissions, 9 accepted, 25 unknown

B: Buggy Blinkers

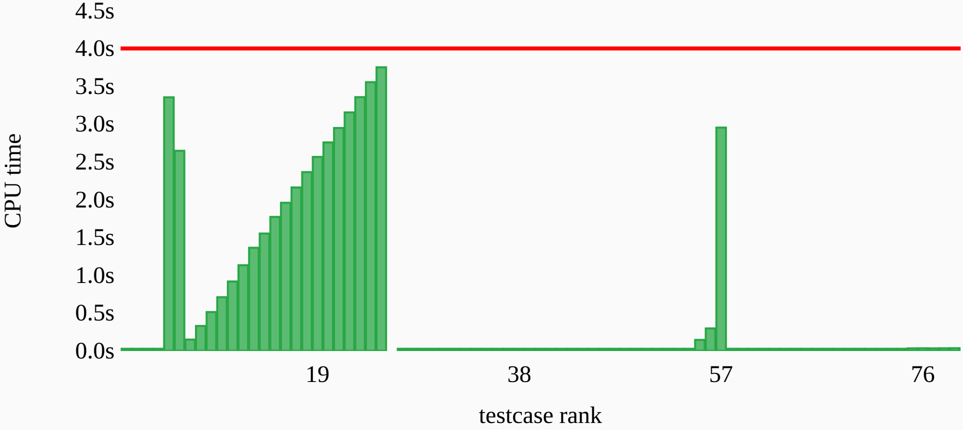
Problem author: Gijs Pennings





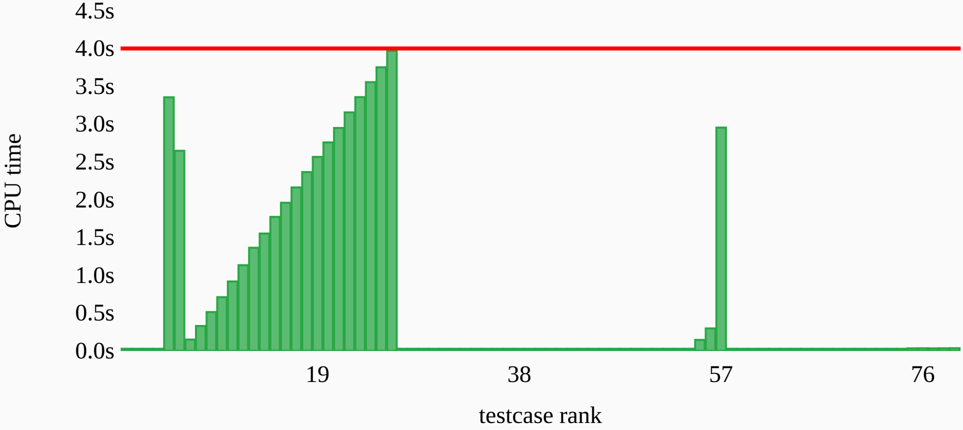
B: Buggy Blinkers

Problem author: Gijs Pennings



B: Buggy Blinkers

Problem author: Gijs Pennings



Only 0.032 seconds to spare!

C: Concurrent Contests

Problem author: Reinier Schmiermann

Problem: Sort contestants into contests such that no-one wants to switch.

C: Concurrent Contests

Problem author: Reinier Schmiermann

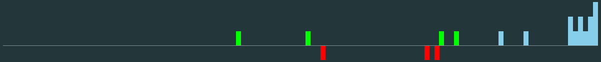
Problem: Sort contestants into contests such that no-one wants to switch.

Solution: A greedy solution works:

1. Sort all contestants by their skill in descending order.
2. Put each contestant in the contest with the highest expected value for them.
3. When done, the resulting distribution of people is such an optimal distribution.

C: Concurrent Contests

Problem author: Reinier Schmiermann



Problem: Sort contestants into contests such that no-one wants to switch.

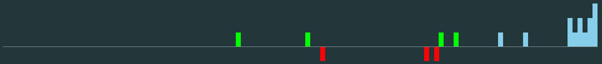
Solution: A greedy solution works:

1. Sort all contestants by their skill in descending order.
2. Put each contestant in the contest with the highest expected value for them.
3. When done, the resulting distribution of people is such an optimal distribution.

Proof sketch: Given a solution, if a person wants to switch, everyone with lower skill also wants to. If someone wants to switch at the end, a contestant with higher skill would have picked a different contest. They didn't, so this must be optimal.

C: Concurrent Contests

Problem author: Reinier Schmiermann



Problem: Sort contestants into contests such that no-one wants to switch.

Solution: A greedy solution works:

1. Sort all contestants by their skill in descending order.
2. Put each contestant in the contest with the highest expected value for them.
3. When done, the resulting distribution of people is such an optimal distribution.

Proof sketch: Given a solution, if a person wants to switch, everyone with lower skill also wants to. If someone wants to switch at the end, a contestant with higher skill would have picked a different contest. They didn't, so this must be optimal.

Pitfall: Floating point numbers: The expected value of joining a contest is given by

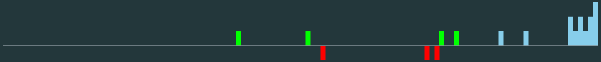
$$\frac{\text{prize} \cdot \text{skill}}{\text{total skill in contest}}$$

Comparing these values naively will lead to floating point errors. Instead, use

$$a/b < c/d \quad \Leftrightarrow \quad a \cdot d < c \cdot b.$$

C: Concurrent Contests

Problem author: Reinier Schmiermann



Problem: Sort contestants into contests such that no-one wants to switch.

Solution: A greedy solution works:

1. Sort all contestants by their skill in descending order.
2. Put each contestant in the contest with the highest expected value for them.
3. When done, the resulting distribution of people is such an optimal distribution.

Proof sketch: Given a solution, if a person wants to switch, everyone with lower skill also wants to. If someone wants to switch at the end, a contestant with higher skill would have picked a different contest. They didn't, so this must be optimal.

Pitfall: Floating point numbers: The expected value of joining a contest is given by

$$\frac{\text{prize} \cdot \text{skill}}{\text{total skill in contest}}$$

Comparing these values naively will lead to floating point errors. Instead, use

$$a/b < c/d \quad \Leftrightarrow \quad a \cdot d < c \cdot b.$$

Running time: $\mathcal{O}(nm)$.

Problem: Sort contestants into contests such that no-one wants to switch.

Solution: A greedy solution works:

1. Sort all contestants by their skill in descending order.
2. Put each contestant in the contest with the highest expected value for them.
3. When done, the resulting distribution of people is such an optimal distribution.

Proof sketch: Given a solution, if a person wants to switch, everyone with lower skill also wants to. If someone wants to switch at the end, a contestant with higher skill would have picked a different contest. They didn't, so this must be optimal.

Pitfall: Floating point numbers: The expected value of joining a contest is given by

$$\frac{\text{prize} \cdot \text{skill}}{\text{total skill in contest}}$$

Comparing these values naively will lead to floating point errors. Instead, use

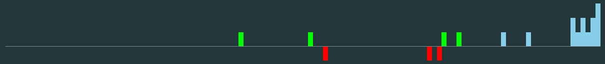
$$a/b < c/d \quad \Leftrightarrow \quad a \cdot d < c \cdot b.$$

Running time: $\mathcal{O}(nm)$.

"Brute force": Alternatively you could repeatedly move people to better spots if you were fast enough.

C: Concurrent Contests

Problem author: Reinier Schmiermann



Problem: Sort contestants into contests such that no-one wants to switch.

Solution: A greedy solution works:

1. Sort all contestants by their skill in descending order.
2. Put each contestant in the contest with the highest expected value for them.
3. When done, the resulting distribution of people is such an optimal distribution.

Proof sketch: Given a solution, if a person wants to switch, everyone with lower skill also wants to. If someone wants to switch at the end, a contestant with higher skill would have picked a different contest. They didn't, so this must be optimal.

Pitfall: Floating point numbers: The expected value of joining a contest is given by

$$\frac{\text{prize} \cdot \text{skill}}{\text{total skill in contest}}$$

Comparing these values naively will lead to floating point errors. Instead, use

$$a/b < c/d \quad \Leftrightarrow \quad a \cdot d < c \cdot b.$$

Running time: $\mathcal{O}(nm)$.

“Brute force”: Alternatively you could repeatedly move people to better spots if you were fast enough.

Statistics: 20 submissions, 4 accepted, 13 unknown

Problem: Given a table number t and menu item m, determine which pinned-up tickets must be flipped to prove the following claim:

$$\forall \text{ pinned-up tickets “Ld”} \quad : \quad (d = t) \to (L = m).$$

D: Disgruntled Diner

Problem author: Wietze Koops

Problem: Given a table number t and menu item m, determine which pinned-up tickets must be flipped to prove the following claim:

$$\forall \text{ pinned-up tickets “Ld”} \quad : \quad (d = t) \rightarrow (L = m).$$

Observation: The claim is false if and only if

$$\exists \text{ pinned-up ticket “Ld”} \quad : \quad (d = t) \wedge (L \neq m).$$

Let's call such tickets *illegal*.

D: Disgruntled Diner

Problem author: Wietze Koops

Claim: $\forall$ pinned-up tickets "Ld" : $(d = t) \to (L = m)$.

Definition: A ticket "Ld" is *illegal* if $(d = t) \wedge (L \neq m)$.

D: Disgruntled Diner

Problem author: Wietze Koops

Claim: $\forall$ pinned-up tickets "Ld" $\quad : \quad (d = t) \rightarrow (L = m)$.

Definition: A ticket "Ld" is *illegal* if $(d = t) \wedge (L \neq m)$.

Solution:

1. *Partition* the computer tickets into sets of legal and illegal tickets. If there are no illegal tickets, we can immediately return "true".
2. Compute a *bipartite matching* between the legal tickets and the pinboard. If this is impossible, the claim is guaranteed to be "false".
3. Now, a matching exists, but any legal ticket on the board could be replaced by an illegal one if the upright side is the same. So, a ticket "x" must be *flipped* if:
   - $x = t$, or
   - there exists an illegal ticket with menu item x.

Claim: $\forall$ pinned-up tickets "Ld" : $(d = t) \rightarrow (L = m)$.

Definition: A ticket "Ld" is *illegal* if $(d = t) \wedge (L \neq m)$.

Solution:

1. *Partition* the computer tickets into sets of legal and illegal tickets. If there are no illegal tickets, we can immediately return "true".
2. Compute a *bipartite matching* between the legal tickets and the pinboard. If this is impossible, the claim is guaranteed to be "false".
3. Now, a matching exists, but any legal ticket on the board could be replaced by an illegal one if the upright side is the same. So, a ticket "x" must be *flipped* if:
   - $x = t$, or
   - there exists an illegal ticket with menu item x.

Running time: Only step 2 is costly. An $\mathcal{O}(n^3)$ flow-based algorithm suffices.

D: Disgruntled Diner

Problem author: Wietze Koops

Claim: $\forall$ pinned-up tickets "Ld" : $(d = t) \to (L = m)$.

Definition: A ticket "Ld" is *illegal* if $(d = t) \wedge (L \neq m)$.

Solution:

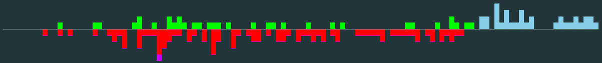
1. *Partition* the computer tickets into sets of legal and illegal tickets. If there are no illegal tickets, we can immediately return "true".
2. Compute a *bipartite matching* between the legal tickets and the pinboard. If this is impossible, the claim is guaranteed to be "false".
3. Now, a matching exists, but any legal ticket on the board could be replaced by an illegal one if the upright side is the same. So, a ticket "x" must be *flipped* if:
   - $x = t$, or
   - there exists an illegal ticket with menu item x.

Running time: Only step 2 is costly. An $\mathcal{O}(n^3)$ flow-based algorithm suffices.

Statistics: 3 submissions, 0 accepted, 3 unknown

E: Extraterrestrial Exploration

Problem author: Wietze Koops

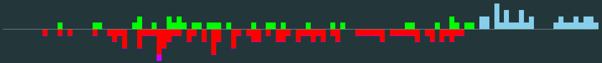


Problem: Given query access to a sorted list of integers $o_1, o_2 \dots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

E: Extraterrestrial Exploration

Problem author: Wietze Koops



Problem: Given query access to a sorted list of integers $o_1, o_2 \ldots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

Naive solution: Check all possible triples and compute the maximum. This is $\mathcal{O}(n^3)$, which is too slow, but more importantly, there are way too few queries to determine the values of all o_i!

Problem: Given query access to a sorted list of integers $o_1, o_2 \ldots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

Observation 1: It is always optimal to include o_1 and o_n. Thus we only need to find y that maximizes

$$\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|} + \sqrt{|o_n - o_1|}.$$

Problem author: Wietze Koops

Problem: Given query access to a sorted list of integers $o_1, o_2 \ldots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

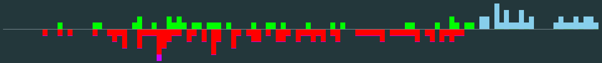
Observation 1: It is always optimal to include o_1 and o_n. Thus we only need to find y that maximizes

$$\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|} + \sqrt{|o_n - o_1|}.$$

Observation 2: The function $\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|}$ is concave and symmetric around $\frac{1}{2}(o_1 + o_n)$. The maximum is attained when o_y is closest to $\frac{1}{2}(o_1 + o_n)$.

E: Extraterrestrial Exploration

Problem author: Wietze Koops



Problem: Given query access to a sorted list of integers $o_1, o_2 \ldots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

Observation 1: It is always optimal to include o_1 and o_n. Thus we only need to find y that maximizes

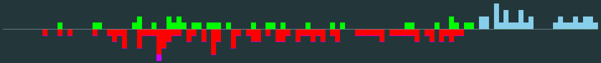
$$\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|} + \sqrt{|o_n - o_1|}.$$

Observation 2: The function $\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|}$ is concave and symmetric around $\frac{1}{2}(o_1 + o_n)$. The maximum is attained when o_y is closest to $\frac{1}{2}(o_1 + o_n)$.

Solution: Query o_1 and o_n, followed by a binary search to find the value o_y closest to $\frac{1}{2}(o_1 + o_n)$.

E: Extraterrestrial Exploration

Problem author: Wietze Koops



Problem: Given query access to a sorted list of integers $o_1, o_2 \dots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

Observation 1: It is always optimal to include o_1 and o_n. Thus we only need to find y that maximizes

$$\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|} + \sqrt{|o_n - o_1|}.$$

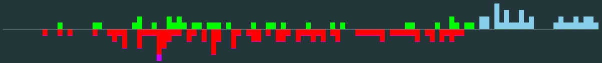
Observation 2: The function $\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|}$ is concave and symmetric around $\frac{1}{2}(o_1 + o_n)$. The maximum is attained when o_y is closest to $\frac{1}{2}(o_1 + o_n)$.

Solution: Query o_1 and o_n, followed by a binary search to find the value o_y closest to $\frac{1}{2}(o_1 + o_n)$.

Alternatively: Find the maximum with ternary search. Need to be somewhat smart with queries to stay within the limit.

E: Extraterrestrial Exploration

Problem author: Wietze Koops



Problem: Given query access to a sorted list of integers $o_1, o_2 \ldots, o_n$, determine x, y, z that maximize

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}.$$

Observation 1: It is always optimal to include o_1 and o_n. Thus we only need to find y that maximizes

$$\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|} + \sqrt{|o_n - o_1|}.$$

Observation 2: The function $\sqrt{|o_1 - o_y|} + \sqrt{|o_y - o_n|}$ is concave and symmetric around $\frac{1}{2}(o_1 + o_n)$. The maximum is attained when o_y is closest to $\frac{1}{2}(o_1 + o_n)$.

Solution: Query o_1 and o_n, followed by a binary search to find the value o_y closest to $\frac{1}{2}(o_1 + o_n)$.

Alternatively: Find the maximum with ternary search. Need to be somewhat smart with queries to stay within the limit.

Pitfall: Make sure to return distinct indices!

Testing tool: There was an issue with the provided testing tool: solutions were compared by computing

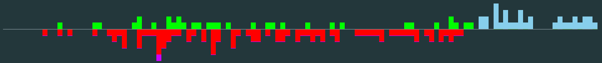
$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}$$

as a floating-point number.

```
Your selection has a value of 7.611767402951557, but the optimal value is 7.611767402951558.
```

E: Extraterrestrial Exploration

Problem author: Wietze Koops



Testing tool: There was an issue with the provided testing tool: solutions were compared by computing

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}$$

as a floating-point number.

```
Your selection has a value of 7.611767402951557, but the optimal value is 7.611767402951558.
```

Fortunately: Judging does not use floating-point comparisons. Instead, check whether there is a middle index that gives a value closer to $\frac{1}{2}(o_1 + o_n)$.

Testing tool: There was an issue with the provided testing tool: solutions were compared by computing

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}$$

as a floating-point number.

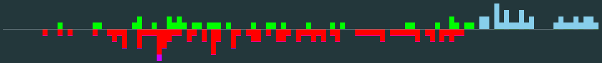
```
Your selection has a value of 7.611767402951557, but the optimal value is 7.611767402951558.
```

Fortunately: Judging does not use floating-point comparisons. Instead, check whether there is a middle index that gives a value closer to $\frac{1}{2}(o_1 + o_n)$.

Doing this in the testing tool would have given away a large part of the solution.

E: Extraterrestrial Exploration

Problem author: Wietze Koops



Testing tool: There was an issue with the provided testing tool: solutions were compared by computing

$$\sqrt{|o_x - o_y|} + \sqrt{|o_y - o_z|} + \sqrt{|o_z - o_x|}$$

as a floating-point number.

```
Your selection has a value of 7.611767402951557, but the optimal value is 7.611767402951558.
```

Fortunately: Judging does not use floating-point comparisons. Instead, check whether there is a middle index that gives a value closer to $\frac{1}{2}(o_1 + o_n)$.

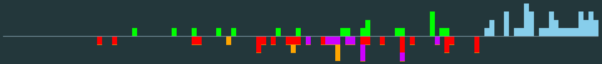
Doing this in the testing tool would have given away a large part of the solution.

Statistics: 159 submissions, 34 accepted, 33 unknown

Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

F: Failing Factory

Problem author: Reinier Schmiermann

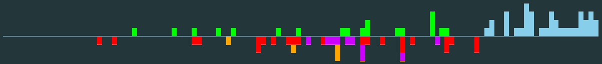


Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

F: Failing Factory

Problem author: Reinier Schmiermann



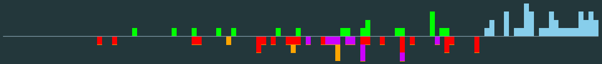
Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

F: Failing Factory

Problem author: Reinier Schmiermann



Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

F: Failing Factory

Problem author: Reinier Schmiermann

Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

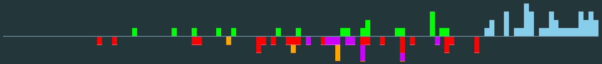
Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

F: Failing Factory



Problem author: Reinier Schmiermann

Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

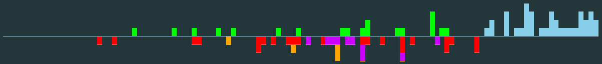
Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

Solution: For each sink component, compute $\prod(1 - p_i)$, and return the maximum.

F: Failing Factory

Problem author: Reinier Schmiermann



Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

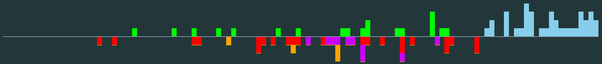
Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

Solution: For each sink component, compute $\prod(1 - p_i)$, and return the maximum.

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

F: Failing Factory

Problem author: Reinier Schmiermann



Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

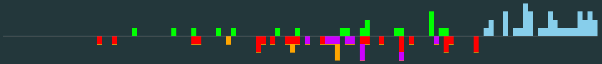
Solution: For each sink component, compute $\prod(1-p_i)$, and return the maximum.

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

Pitfalls: Do not confuse min and max.

F: Failing Factory

Problem author: Reinier Schmiermann



Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

Solution: For each sink component, compute $\prod(1 - p_i)$, and return the maximum.

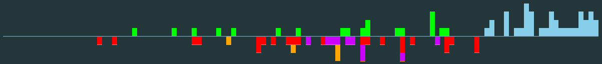
Running time: $\mathcal{O}(n)$.

Pitfalls: Do not confuse min and max.

Pitfalls: Do not confuse p and $1 - p$.

F: Failing Factory

Problem author: Reinier Schmiermann



Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

Solution: For each sink component, compute $\prod(1 - p_i)$, and return the maximum.

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

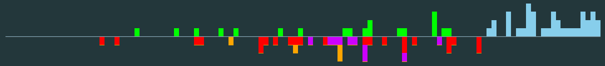
Pitfalls: Do not confuse min and max.

Pitfalls: Do not confuse p and $1 - p$.

Pitfalls: Print output with sufficient precision (e.g. using `setprecision(10)`).

F: Failing Factory

Problem author: Reinier Schmiermann



Problem: Given is a graph of the dependencies between steps in a factory. Each step independently fails with some probability p_i. Find the maximum probability that a step and all its dependencies do not fail.

Naive solution: For each step, multiply the success probabilities of all its dependencies, using DFS. $\mathcal{O}(n^2)$ is too slow!

Insight 1: Within a strongly connected component, all steps have the same failure probability.

Insight 2: We should look for a SCC without external dependencies. (So a sink in the collapsed graph).

Solution: Use Tarjan's or Kosaraju's algorithm to find strongly connected components.

Solution: For each sink component, compute $\prod(1 - p_i)$, and return the maximum.

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

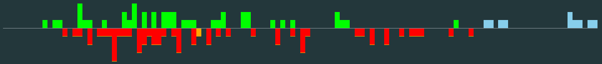
Pitfalls: Do not confuse min and max.

Pitfalls: Do not confuse p and $1 - p$.

Pitfalls: Print output with sufficient precision (e.g. using `setprecision(10)`).

Statistics: 92 submissions, 19 accepted, 36 unknown

G: Grocery Greed

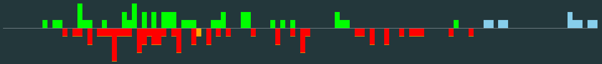


Problem author: Jorke de Vlas

Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

G: Grocery Greed

Problem author: Jorke de Vlas

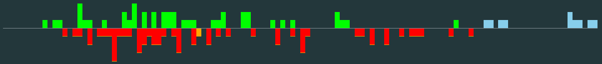


Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Observation 1: Increasing or decreasing a price by a multiple of 5 cents will always change the final answer by this same amount.

G: Grocery Greed

Problem author: Jorke de Vlas



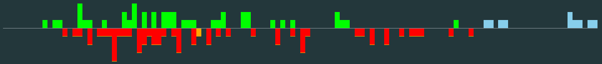
Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Observation 1: Increasing or decreasing a price by a multiple of 5 cents will always change the final answer by this same amount.

- We may reduce all prices modulo 5 cents, as long as we add the differences to the final result.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

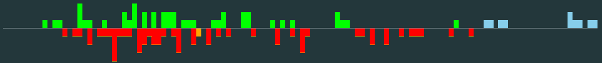
Observation 1: Increasing or decreasing a price by a multiple of 5 cents will always change the final answer by this same amount.

- We may reduce all prices modulo 5 cents, as long as we add the differences to the final result.

Observation 2: It is always optimal to put prices of 0, 1 or 2 cents in their own group and round this down to 0 cents.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Observation 1: Increasing or decreasing a price by a multiple of 5 cents will always change the final answer by this same amount.

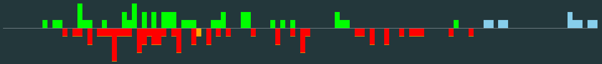
- We may reduce all prices modulo 5 cents, as long as we add the differences to the final result.

Observation 2: It is always optimal to put prices of 0, 1 or 2 cents in their own group and round this down to 0 cents.

Remaining case: All prices are 3 or 4 cents.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Remaining case: All prices are 3 or 4 cents.

G: Grocery Greed

Problem author: Jorke de Vlas

Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

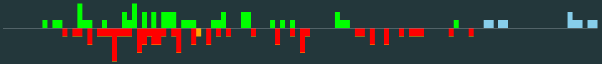
Remaining case: All prices are 3 or 4 cents.

Solution:

- As long as there is both a price of 3 cents and a price of 4 cents, put them together in a group and round to 5 cents.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

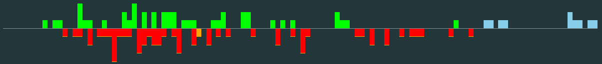
Remaining case: All prices are 3 or 4 cents.

Solution:

- As long as there is both a price of 3 cents and a price of 4 cents, put them together in a group and round to 5 cents.
- If only prices of 3 cents remain: make as many pairs as possible and round them down to 5 cents. If a single price of 3 cents remains, do not round it.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

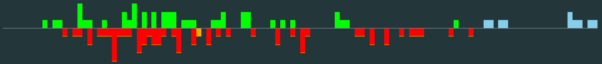
Remaining case: All prices are 3 or 4 cents.

Solution:

- As long as there is both a price of 3 cents and a price of 4 cents, put them together in a group and round to 5 cents.
- If only prices of 3 cents remain: make as many pairs as possible and round them down to 5 cents. If a single price of 3 cents remains, do not round it.
- If only prices of 4 cents remain: make as many triples as possible and round them down to 10 cents. If one or two prices of 4 cents remain, do not round them.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Remaining case: All prices are 3 or 4 cents.

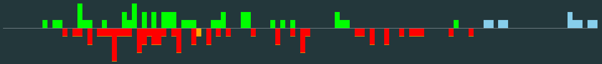
Solution:

- As long as there is both a price of 3 cents and a price of 4 cents, put them together in a group and round to 5 cents.
- If only prices of 3 cents remain: make as many pairs as possible and round them down to 5 cents. If a single price of 3 cents remains, do not round it.
- If only prices of 4 cents remain: make as many triples as possible and round them down to 10 cents. If one or two prices of 4 cents remain, do not round them.

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

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Remaining case: All prices are 3 or 4 cents.

Solution:

- As long as there is both a price of 3 cents and a price of 4 cents, put them together in a group and round to 5 cents.
- If only prices of 3 cents remain: make as many pairs as possible and round them down to 5 cents. If a single price of 3 cents remains, do not round it.
- If only prices of 4 cents remain: make as many triples as possible and round them down to 10 cents. If one or two prices of 4 cents remain, do not round them.

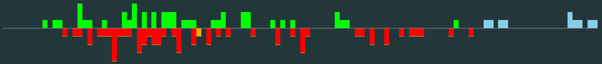
Running time: $\mathcal{O}(n)$.

Pitfall: Be careful converting between floating point numbers and integers!

- Casting `100*x` (which is `float`) to `int` is flooring, so add `0.5` or use `round()`.
- Alternatively, skip the decimal point when parsing the input values to `int`.

G: Grocery Greed

Problem author: Jorke de Vlas



Problem: Given a list of prices, divide them into groups and decide for each group whether to round the price of the group to a multiple of 5 cents, to minimize the total price.

Remaining case: All prices are 3 or 4 cents.

Solution:

- As long as there is both a price of 3 cents and a price of 4 cents, put them together in a group and round to 5 cents.
- If only prices of 3 cents remain: make as many pairs as possible and round them down to 5 cents. If a single price of 3 cents remains, do not round it.
- If only prices of 4 cents remain: make as many triples as possible and round them down to 10 cents. If one or two prices of 4 cents remain, do not round them.

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

Pitfall: Be careful converting between floating point numbers and integers!

- Casting `100*x` (which is `float`) to `int` is flooring, so add `0.5` or use `round()`.
- Alternatively, skip the decimal point when parsing the input values to `int`.

Statistics: 111 submissions, 44 accepted, 10 unknown

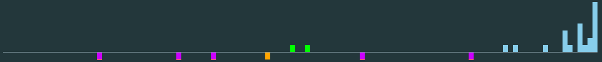
H: Horse Habitat

Problem author: Mike de Vries

Problem: Given a square grid with r rows and c columns, each square being either '.' or '#'. Determine for each $1 \leq w \leq c$ and $1 \leq h \leq r$ the number of $w \times h$ rectangles in the grid with only '.'.

H: Horse Habitat

Problem author: Mike de Vries



Problem: Given a square grid with r rows and c columns, each square being either '.' or '#'. Determine for each $1 \leq w \leq c$ and $1 \leq h \leq r$ the number of $w \times h$ rectangles in the grid with only '.'.

Observation: Going from top to bottom, we can determine for every square (i, j) the distance $d(i, j)$ to the first '#' above it, assuming out of bounds is '#'. Then a $w \times h$ rectangle fits somewhere if and only if the bottom row consists only of values that are at least h.

H: Horse Habitat

Problem author: Mike de Vries

Problem: Given a square grid with r rows and c columns, each square being either '.' or '#'. Determine for each $1 \leq w \leq c$ and $1 \leq h \leq r$ the number of $w \times h$ rectangles in the grid with only '.'.

Observation: Going from top to bottom, we can determine for every square (i, j) the distance $d(i, j)$ to the first '#' above it, assuming out of bounds is '#'. Then a $w \times h$ rectangle fits somewhere if and only if the bottom row consists only of values that are at least h.

Solution: For each square (i, j) determine for each $k \geq j$ the smallest value v of $d(i, t)$ for $t = j, \ldots, k$ and report a $(k - j + 1) \times v$ rectangle. The number of $w \times h$ rectangles is then the total number of reported $w \times v$ rectangles for $v \geq h$.

H: Horse Habitat

Problem author: Mike de Vries

Problem: Given a square grid with r rows and c columns, each square being either '.' or '#'. Determine for each $1 \leq w \leq c$ and $1 \leq h \leq r$ the number of $w \times h$ rectangles in the grid with only '.'.

Observation: Going from top to bottom, we can determine for every square (i, j) the distance $d(i, j)$ to the first '#' above it, assuming out of bounds is '#'. Then a $w \times h$ rectangle fits somewhere if and only if the bottom row consists only of values that are at least h.

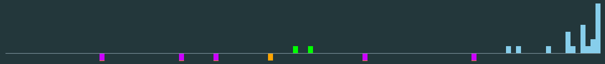
Solution: For each square (i, j) determine for each $k \geq j$ the smallest value v of $d(i, t)$ for $t = j, \ldots, k$ and report a $(k - j + 1) \times v$ rectangle. The number of $w \times h$ rectangles is then the total number of reported $w \times v$ rectangles for $v \geq h$.

Running time: For $n = rc$: $\mathcal{O}(rc^2) = \mathcal{O}(n\sqrt{n})$ after possibly transposing to make sure $c \leq r$. Too slow!

New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

H: Horse Habitat

Problem author: Mike de Vries



New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Task: In order to do this efficiently, we need to keep track of the total number of maximal intervals of all possible lengths.

New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Task: In order to do this efficiently, we need to keep track of the total number of maximal intervals of all possible lengths.

Solution: For each row we can keep track for each boundary of a maximal interval the other boundary of that interval, or itself if the interval is of length one. For each new square we can initialize a new interval of length one and then potentially merge with the intervals on the left and right if they exist. Consider the test case below:

New plan: For h from r down to 1, we can keep track of which squares (i,j) have $d(i,j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Task: In order to do this efficiently, we need to keep track of the total number of maximal intervals of all possible lengths.

Solution: For each row we can keep track for each boundary of a maximal interval the other boundary of that interval, or itself if the interval is of length one. For each new square we can initialize a new interval of length one and then potentially merge with the intervals on the left and right if they exist. Consider the test case below:

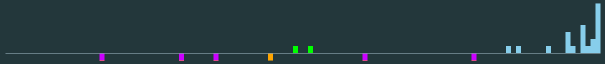
```
#..##
#..#.
#....
```

0	3	3	1	2

$h = 3$

H: Horse Habitat

Problem author: Mike de Vries



New plan: For h from r down to 1, we can keep track of which squares (i,j) have $d(i,j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

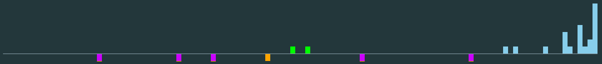
Task: In order to do this efficiently, we need to keep track of the total number of maximal intervals of all possible lengths.

Solution: For each row we can keep track for each boundary of a maximal interval the other boundary of that interval, or itself if the interval is of length one. For each new square we can initialize a new interval of length one and then potentially merge with the intervals on the left and right if they exist. Consider the test case below:

```
#..##
#..#.
#....
```

0	3	3	1	2

$h = 2$



New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Task: In order to do this efficiently, we need to keep track of the total number of maximal intervals of all possible lengths.

Solution: For each row we can keep track for each boundary of a maximal interval the other boundary of that interval, or itself if the interval is of length one. For each new square we can initialize a new interval of length one and then potentially merge with the intervals on the left and right if they exist. Consider the test case below:

```
#..##
#..#.
#....
```

0	3	3	1	2

$h = 1$

H: Horse Habitat

Problem author: Mike de Vries

New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Solved: We can keep track of the total number of maximal intervals of all possible lengths.

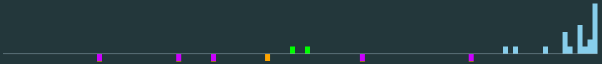
New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Solved: We can keep track of the total number of maximal intervals of all possible lengths.

Calculation: After adding all squares with values of at least h, let I_w be the number of maximal intervals of length w. The total number of $w \times h$ rectangles can then be calculated as $R_w = I_w + 2I_{w+1} + \ldots$.

H: Horse Habitat

Problem author: Mike de Vries



New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Solved: We can keep track of the total number of maximal intervals of all possible lengths.

Calculation: After adding all squares with values of at least h, let I_w be the number of maximal intervals of length w. The total number of $w \times h$ rectangles can then be calculated as $R_w = I_w + 2I_{w+1} + \ldots$.

Finish: Introducing the value $C_w = I_w + I_{w+1} + \ldots$ we get $C_w = I_w + C_{w+1}$ and $R_w = C_w + R_{w+1}$. We can thus calculate C_w and R_w recursively for all h.

H: Horse Habitat

Problem author: Mike de Vries

New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Solved: We can keep track of the total number of maximal intervals of all possible lengths.

Calculation: After adding all squares with values of at least h, let I_w be the number of maximal intervals of length w. The total number of $w \times h$ rectangles can then be calculated as $R_w = I_w + 2I_{w+1} + \dots$.

Finish: Introducing the value $C_w = I_w + I_{w+1} + \dots$ we get $C_w = I_w + C_{w+1}$ and $R_w = C_w + R_{w+1}$. We can thus calculate C_w and R_w recursively for all h.

Running time: Linear time: $\mathcal{O}(rc)$

H: Horse Habitat

Problem author: Mike de Vries

New plan: For h from r down to 1, we can keep track of which squares (i, j) have $d(i, j) \geq h$. We can then determine the number of $w \times h$ rectangles for any w by counting the number of intervals of length w in each row.

Solved: We can keep track of the total number of maximal intervals of all possible lengths.

Calculation: After adding all squares with values of at least h, let I_w be the number of maximal intervals of length w. The total number of $w \times h$ rectangles can then be calculated as $R_w = I_w + 2I_{w+1} + \ldots$.

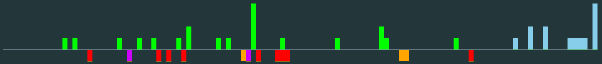
Finish: Introducing the value $C_w = I_w + I_{w+1} + \ldots$ we get $C_w = I_w + C_{w+1}$ and $R_w = C_w + R_{w+1}$. We can thus calculate C_w and R_w recursively for all h.

Running time: Linear time: $\mathcal{O}(rc)$

Statistics: 29 submissions, 2 accepted, 21 unknown

I: Interrail Pass

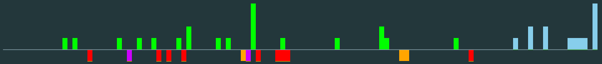
Problem author: Ragnar Groot Koerkamp



Problem: Pay trips on n days $t_i \in \{0, \ldots, T\}$. The fare for the ith trip is f_i. Instead of paying the fare you can use a (multi-ride) *pass*. There are k types of pass, the jth has cost c_j and lasts for a period of p_j days, during which it covers the first d_j trips.

I: Interrail Pass

Problem author: Ragnar Groot Koerkamp

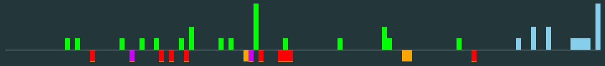


Problem: Pay trips on n days $t_i \in \{0, \ldots, T\}$. The fare for the ith trip is f_i. Instead of paying the fare you can use a (multi-ride) *pass*. There are k types of pass, the jth has cost c_j and lasts for a period of p_j days, during which it covers the first d_j trips.

First step: Focus on the trip dates $t_1, \ldots, t_n$ (rather than $\{0, \ldots, T\}$). Useful to understand the process 'backwards': "if I pay the ith trip (on day t_i) with the jth pass, then ...."

I: Interrail Pass

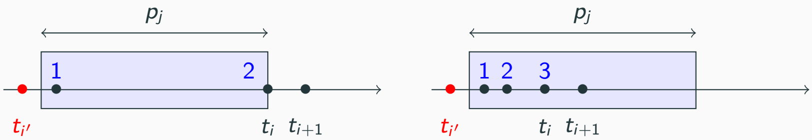
Problem author: Ragnar Groot Koerkamp



Problem: Pay trips on n days $t_i \in \{0, \dots, T\}$. The fare for the ith trip is f_i. Instead of paying the fare you can use a (multi-ride) *pass*. There are k types of pass, the jth has cost c_j and lasts for a period of p_j days, during which it covers the first d_j trips.

First step: Focus on the trip dates $t_1, \dots, t_n$ (rather than $\{0, \dots, T\}$). Useful to understand the process 'backwards': "if I pay the ith trip (on day t_i) with the jth pass, then ...."

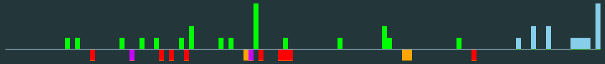
Can assume that today is the *last* travel day covered by the pass, either because period p_j ran out before t_{i+1} or because the pass ran out of days d_j. E.g., for $d_j = 3$:



The index of the first travel day not covered by jth pass is therefore the largest $i' \geq 1$ such that $i' \leq i - d_j$ or $t_{i'} \leq t_i - p_j$.

I: Interrail Pass

Problem author: Ragnar Groot Koerkamp



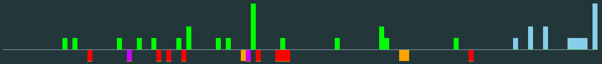
Definition: For $t \in \{1, \dots, T\}$, let $\mathrm{prev}(t)$ be the index of latest trip before day t, formally

$$\mathrm{prev}(t) = \max\{\, i \colon t_i \le t \,\}\,.$$

This can be evaluated in time $\mathcal{O}(\log n)$ by binary search in $(t_1, \dots, t_n)$ or in constant time with $\mathcal{O}(T)$ preprocessing by tabulating $\mathrm{prev}(t)$ for every t.

I: Interrail Pass

Problem author: Ragnar Groot Koerkamp



Definition: For $t \in \{1, \ldots, T\}$, let $\mathrm{prev}(t)$ be the index of latest trip before day t, formally

$$\mathrm{prev}(t) = \max\{\, i \colon t_i \leq t \,\}\,.$$

This can be evaluated in time $\mathcal{O}(\log n)$ by binary search in $(t_1, \ldots, t_n)$ or in constant time with $\mathcal{O}(T)$ preprocessing by tabulating $\mathrm{prev}(t)$ for every t.

Recurrence: Let OPT(i) be the optimum cost for the first i trips. Then, for $i > 0$,

$$\mathrm{OPT}(i) = \min \begin{cases} f_i + \mathrm{OPT}(i-1)\,, & (\textit{pay regular fare}) \\ \min\limits_{1 \leq j \leq k} \left\{ c_j + \mathrm{OPT}(\max(i - d_j, \mathrm{prev}(t_i - p_j))) \right\} & (j\textit{th pass expires today}) \end{cases}$$

Definition: For $t \in \{1, \ldots, T\}$, let $\operatorname{prev}(t)$ be the index of latest trip before day t, formally

$$\operatorname{prev}(t) = \max\{\, i : t_i \le t \,\}.$$

This can be evaluated in time $\mathcal{O}(\log n)$ by binary search in $(t_1, \ldots, t_n)$ or in constant time with $\mathcal{O}(T)$ preprocessing by tabulating $\operatorname{prev}(t)$ for every t.

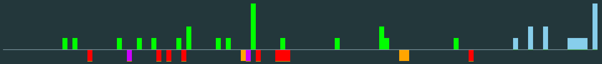
Recurrence: Let OPT(i) be the optimum cost for the first i trips. Then, for $i > 0$,

$$\mathrm{OPT}(i) = \min \begin{cases} f_i + \mathrm{OPT}(i-1)\,, & (\textit{pay regular fare}) \\ \min\limits_{1 \le j \le k} \left\{ c_j + \mathrm{OPT}(\max(i - d_j, \operatorname{prev}(t_i - p_j))) \right\} & (j\textit{th pass expires today}) \end{cases}$$

Solution: Implement using dynamic programming / memoization.

I: Interrail Pass

Problem author: Ragnar Groot Koerkamp



Definition: For $t \in \{1, \dots, T\}$, let prev(t) be the index of latest trip before day t, formally

$$\operatorname{prev}(t) = \max\{\, i\colon\ t_i \leq t \,\}\,.$$

This can be evaluated in time $\mathcal{O}(\log n)$ by binary search in $(t_1, \dots, t_n)$ or in constant time with $\mathcal{O}(T)$ preprocessing by tabulating prev(t) for every t.

Recurrence: Let OPT(i) be the optimum cost for the first i trips. Then, for $i > 0$,

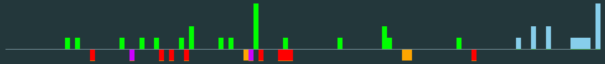
$$\operatorname{OPT}(i) = \min \begin{cases} f_i + \operatorname{OPT}(i-1)\,, & (\textit{pay regular fare}) \\ \min\limits_{1 \leq j \leq k} \left\{ c_j + \operatorname{OPT}(\max(i - d_j, \operatorname{prev}(t_i - p_j))) \right\} & (j\textit{th pass expires today}) \end{cases}$$

Solution: Implement using dynamic programming / memoization.

Running time: $\mathcal{O}(nk \log n)$ or $\mathcal{O}(nk + T)$.

I: Interrail Pass

Problem author: Ragnar Groot Koerkamp



Definition: For $t \in \{1, \ldots, T\}$, let $\operatorname{prev}(t)$ be the index of latest trip before day t, formally

$$\operatorname{prev}(t) = \max\{\, i \colon t_i \leq t \,\}\,.$$

This can be evaluated in time $\mathcal{O}(\log n)$ by binary search in $(t_1, \ldots, t_n)$ or in constant time with $\mathcal{O}(T)$ preprocessing by tabulating $\operatorname{prev}(t)$ for every t.

Recurrence: Let OPT(i) be the optimum cost for the first i trips. Then, for $i > 0$,

$$\operatorname{OPT}(i) = \min \begin{cases} f_i + \operatorname{OPT}(i-1)\,, & \textit{(pay regular fare)} \\ \min\limits_{1 \leq j \leq k} \bigl\{ c_j + \operatorname{OPT}(\max(i - d_j, \operatorname{prev}(t_i - p_j))) \bigr\} & \textit{(jth pass expires today)} \end{cases}$$

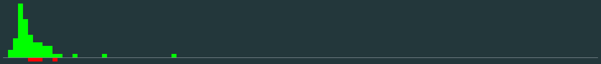
Solution: Implement using dynamic programming / memoization.

Running time: $\mathcal{O}(nk \log n)$ or $\mathcal{O}(nk + T)$.

Statistics: 47 submissions, 20 accepted, 13 unknown

J: Jumbled Scoreboards

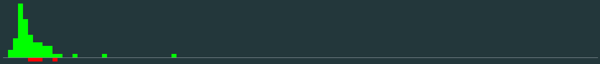
Problem author: Lammert Westerdijk



Problem: Given a list of scoreboards, determine whether they are *chronological*: i.e., if these scoreboards can occur in this order in a single match.

J: Jumbled Scoreboards

Problem author: Lammert Westerdijk

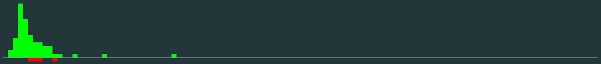


Problem: Given a list of scoreboards, determine whether they are *chronological*: i.e., if these scoreboards can occur in this order in a single match.

Observation: The list of scoreboards is chronological if and only if a list of scores is non-decreasing for both teams.

J: Jumbled Scoreboards

Problem author: Lammert Westerdijk



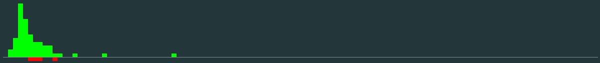
Problem: Given a list of scoreboards, determine whether they are *chronological*: i.e., if these scoreboards can occur in this order in a single match.

Observation: The list of scoreboards is chronological if and only if a list of scores is non-decreasing for both teams.

Solution: For each list, check whether each pair of consecutive scores is non-decreasing.

J: Jumbled Scoreboards

Problem author: Lammert Westerdijk



Problem: Given a list of scoreboards, determine whether they are *chronological*: i.e., if these scoreboards can occur in this order in a single match.

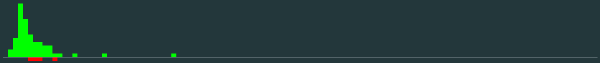
Observation: The list of scoreboards is chronological if and only if a list of scores is non-decreasing for both teams.

Solution: For each list, check whether each pair of consecutive scores is non-decreasing.

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

J: Jumbled Scoreboards

Problem author: Lammert Westerdijk



Problem: Given a list of scoreboards, determine whether they are *chronological*: i.e., if these scoreboards can occur in this order in a single match.

Observation: The list of scoreboards is chronological if and only if a list of scores is non-decreasing for both teams.

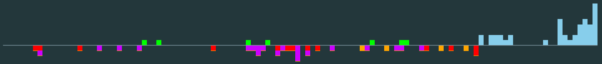
Solution: For each list, check whether each pair of consecutive scores is non-decreasing.

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

Statistics: 60 submissions, 56 accepted

K: Karaoke Compression

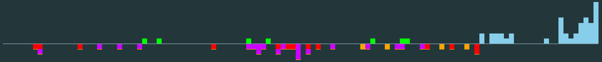
Problem author: Jorke de Vlas



Problem: Compress a string s by replacing all occurrences of a chosen substring by a single character, minimizing the total length of the compressed string and the replaced substring.

K: Karaoke Compression

Problem author: Jorke de Vlas

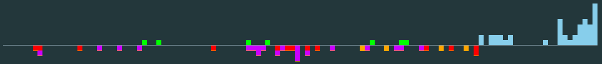


Problem: Compress a string s by replacing all occurrences of a chosen substring by a single character, minimizing the total length of the compressed string and the replaced substring.

Naive solution: For every substring, count the non-overlapping occurrences in s. Running time: $\mathcal{O}(n^4)$, or $\mathcal{O}(n^3)$ with KMP.

K: Karaoke Compression

Problem author: Jorke de Vlas



Problem: Compress a string s by replacing all occurrences of a chosen substring by a single character, minimizing the total length of the compressed string and the replaced substring.

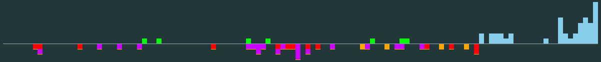
Naive solution: For every substring, count the non-overlapping occurrences in s. Running time: $\mathcal{O}(n^4)$, or $\mathcal{O}(n^3)$ with KMP.

Actual solution: Use rolling hashes to quickly count the occurrences of every substring.

- Define $H(c_i c_{i+1} \dots c_j) = c_i b^0 + c_{i+1} b^1 + \cdots c_j b^{j-i} \mod M$, where we identify every character with an integer and b and M are fixed integers.
- Note that $H(c_i c_{i+1} \dots c_j) = H(c_i c_{i+1} \dots c_{j-1}) + c_j b^{j-i} = c_0 + H(c_1 \dots c_j) \cdot b \mod M$, using which we can compute all hashes in $\mathcal{O}(n^2)$ time.
- Walk through the string, for every hash keeping track of its last occurrence and the number of non-overlapping occurrences found so far. This is also $\mathcal{O}(n^2)$.

K: Karaoke Compression

Problem author: Jorke de Vlas



Problem: Compress a string s by replacing all occurrences of a chosen substring by a single character, minimizing the total length of the compressed string and the replaced substring.

Naive solution: For every substring, count the non-overlapping occurrences in s. Running time: $\mathcal{O}(n^4)$, or $\mathcal{O}(n^3)$ with KMP.

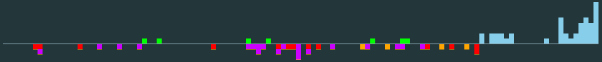
Actual solution: Use rolling hashes to quickly count the occurrences of every substring.

- Define $H(c_i c_{i+1} \dots c_j) = c_i b^0 + c_{i+1} b^1 + \cdots c_j b^{j-i} \mod M$, where we identify every character with an integer and b and M are fixed integers.
- Note that $H(c_i c_{i+1} \dots c_j) = H(c_i c_{i+1} \dots c_{j-1}) + c_j b^{j-i} = c_0 + H(c_1 \dots c_j) \cdot b \mod M$, using which we can compute all hashes in $\mathcal{O}(n^2)$ time.
- Walk through the string, for every hash keeping track of its last occurrence and the number of non-overlapping occurrences found so far. This is also $\mathcal{O}(n^2)$.

Pitfall: Hash collisions. There are $> 10^7$ substrings, so if M has less than 14 digits you will get collisions (birthday paradox).

K: Karaoke Compression

Problem author: Jorke de Vlas



Problem: Compress a string s by replacing all occurrences of a chosen substring by a single character, minimizing the total length of the compressed string and the replaced substring.

Naive solution: For every substring, count the non-overlapping occurrences in s. Running time: $\mathcal{O}(n^4)$, or $\mathcal{O}(n^3)$ with KMP.

Actual solution: Use rolling hashes to quickly count the occurrences of every substring.

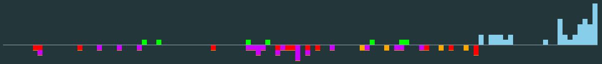
- Define $H(c_i c_{i+1} \dots c_j) = c_i b^0 + c_{i+1} b^1 + \cdots c_j b^{j-i} \mod M$, where we identify every character with an integer and b and M are fixed integers.
- Note that $H(c_i c_{i+1} \dots c_j) = H(c_i c_{i+1} \dots c_{j-1}) + c_j b^{j-i} = c_0 + H(c_1 \dots c_j) \cdot b \mod M$, using which we can compute all hashes in $\mathcal{O}(n^2)$ time.
- Walk through the string, for every hash keeping track of its last occurrence and the number of non-overlapping occurrences found so far. This is also $\mathcal{O}(n^2)$.

Pitfall: Hash collisions. There are $> 10^7$ substrings, so if M has less than 14 digits you will get collisions (birthday paradox).

Note: There are also solutions using DP, divide and conquer, suffix arrays, or the z-function.

K: Karaoke Compression

Problem author: Jorke de Vlas



Problem: Compress a string s by replacing all occurrences of a chosen substring by a single character, minimizing the total length of the compressed string and the replaced substring.

Naive solution: For every substring, count the non-overlapping occurrences in s. Running time: $\mathcal{O}(n^4)$, or $\mathcal{O}(n^3)$ with KMP.

Actual solution: Use rolling hashes to quickly count the occurrences of every substring.

- Define $H(c_i c_{i+1} \dots c_j) = c_i b^0 + c_{i+1} b^1 + \cdots c_j b^{j-i} \mod M$, where we identify every character with an integer and b and M are fixed integers.
- Note that $H(c_i c_{i+1} \dots c_j) = H(c_i c_{i+1} \dots c_{j-1}) + c_j b^{j-i} = c_0 + H(c_1 \dots c_j) \cdot b \mod M$, using which we can compute all hashes in $\mathcal{O}(n^2)$ time.
- Walk through the string, for every hash keeping track of its last occurrence and the number of non-overlapping occurrences found so far. This is also $\mathcal{O}(n^2)$.

Pitfall: Hash collisions. There are $> 10^7$ substrings, so if M has less than 14 digits you will get collisions (birthday paradox).

Note: There are also solutions using DP, divide and conquer, suffix arrays, or the z-function.

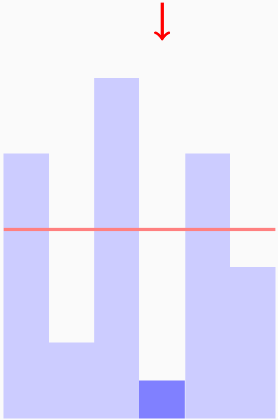
Statistics: 88 submissions, 7 accepted, 43 unknown

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

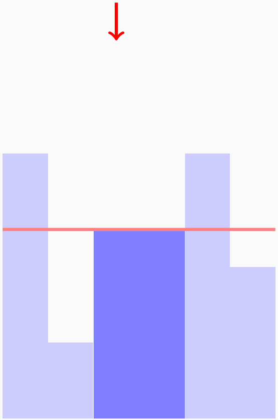
Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

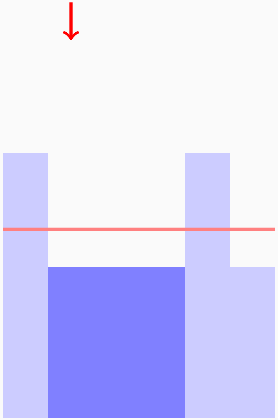
Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

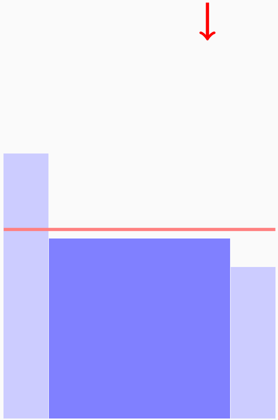
Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

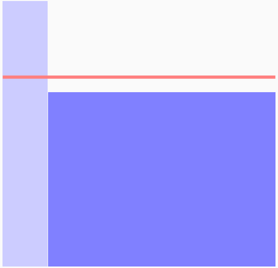
Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

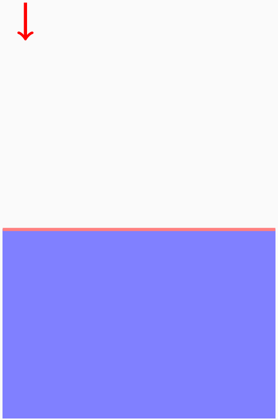
Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.

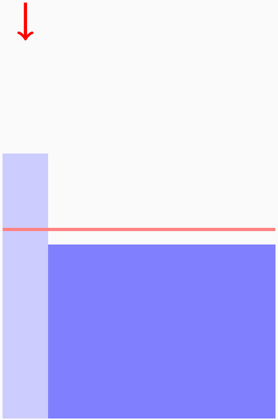
L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.



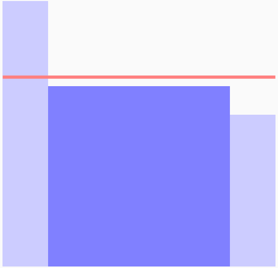
L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.



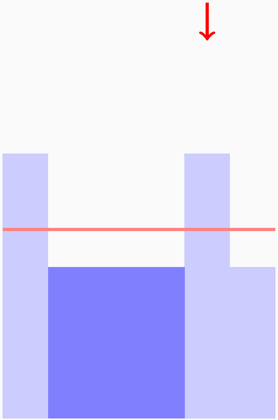
L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

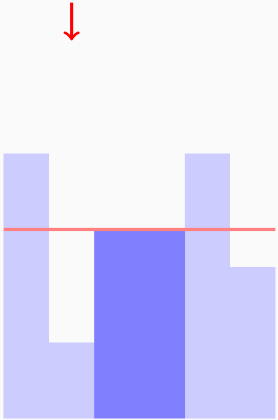
Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.



Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.



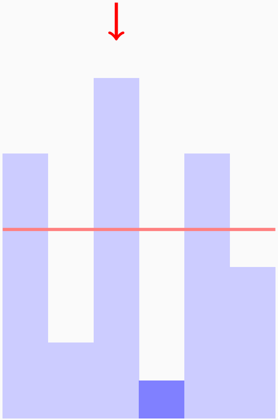
L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

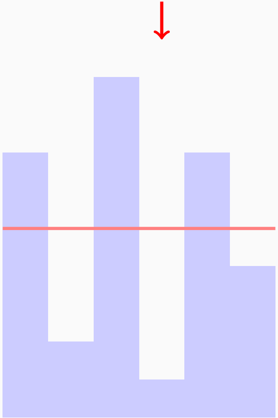
Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.



Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.

Simplification: Subtract A from each element.

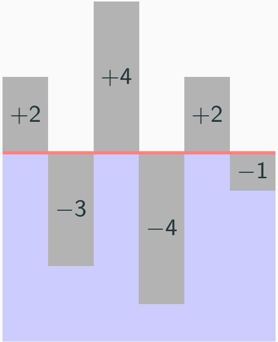
Problem: Given n integers with average A, pick an element and repeatedly add an element left or right such that the average of the chosen range is always $\leq A$.

Simplification: Consider the procedure in reverse.

Problem: Given n integers with average A, repeatedly **delete** an element left or right such that the average is always $\leq A$.

Simplification: Subtract A from each element.

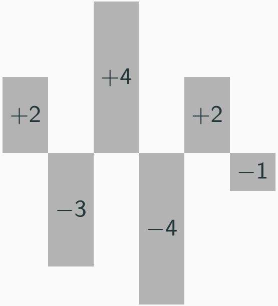
Problem: Given n integers, repeatedly delete elements left or right such that the **sum** is always ≤ 0.



L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

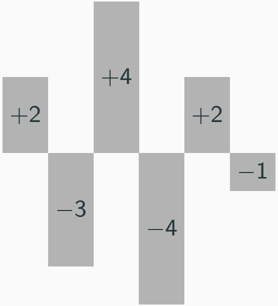
Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.



Statistics: 54 submissions, 0 accepted, 31 unknown

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.



Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

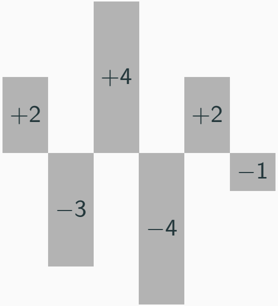
Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.



Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

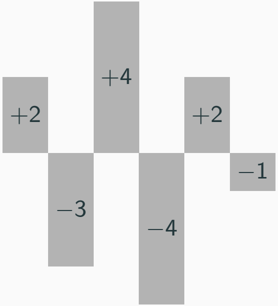
Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.



Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

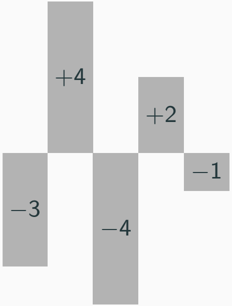
Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.



Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

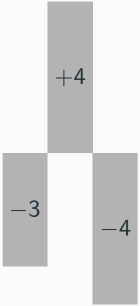
Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix “never hurts”, i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.



Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.

-4

Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.

Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.

Solution: Repeatedly delete the shortest valid non-negative prefix until none remain, then do the same with suffixes, alternating between them. If no such prefix or suffix exists at some point, the answer is "`impossible`".

Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.

Solution: Repeatedly delete the shortest valid non-negative prefix until none remain, then do the same with suffixes, alternating between them. If no such prefix or suffix exists at some point, the answer is "`impossible`".

Complexity: $O(n)$.

Statistics: 54 submissions, 0 accepted, 31 unknown

L: Levelling Locks

Problem author: Ragnar Groot Koerkamp

Problem: Given n integers, repeatedly delete elements left or right such that the sum is always ≤ 0.

Definition: A *valid* prefix is a prefix such that when deleted one by one, the sum in the remaining array is always ≤ 0.

Insight: Deleting the *shortest* valid non-negative prefix or suffix "never hurts", i.e., is optimal.

Intuition: After taking a prefix with positive sum, we can take every suffix we could have taken before.

Solution: Repeatedly delete the shortest valid non-negative prefix until none remain, then do the same with suffixes, alternating between them. If no such prefix or suffix exists at some point, the answer is "`impossible`".

Complexity: $O(n)$.

Pitfall: Floating point imprecision. Use integers or resort to fractions.

Statistics: 54 submissions, 0 accepted, 31 unknown

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

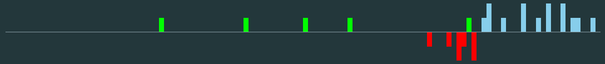
Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

M: Museum Visit

Problem author: Tobias Roehr



Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

Running time: $\mathcal{O}(n^2)$, which is too slow.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

Running time: $\mathcal{O}(n^2)$, which is too slow.

Faster Solution: Use a segment tree to maintain the DP values.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

Running time: $\mathcal{O}(n^2)$, which is too slow.

Faster Solution: Use a segment tree to maintain the DP values.

Running time: $\mathcal{O}(n \log n)$, which is fast enough!

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

Running time: $\mathcal{O}(n^2)$, which is too slow.

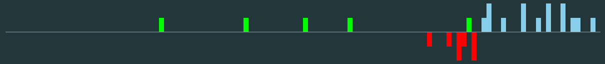
Faster Solution: Use a segment tree to maintain the DP values.

Running time: $\mathcal{O}(n \log n)$, which is fast enough!

Easier Solution: Use a heap to maintain the DP values.

M: Museum Visit

Problem author: Tobias Roehr



Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

Running time: $\mathcal{O}(n^2)$, which is too slow.

Faster Solution: Use a segment tree to maintain the DP values.

Running time: $\mathcal{O}(n \log n)$, which is fast enough!

Easier Solution: Use a heap to maintain the DP values.

Fastest Solution: Use a deque to maintain the pareto front of $(i, \mathrm{dp}[\mathrm{i}])$.

M: Museum Visit

Problem author: Tobias Roehr

Problem: Given intervals and costs associated with each integer, find the minimum cost of a subset of integers that hits all intervals.

Observation: If one interval is contained in another one, we can ignore the larger interval, as it will always be hit when the smaller one is.

Solution: Dynamic programming! Let $\mathrm{dp}[i]$ be the minimum cost of a set that includes i and hits all intervals ending before i.

Recurrence: $\mathrm{dp}[i] = \mathrm{cost}[i] + \min_{j \in [k_i, i)} \mathrm{dp}[j]$ with k_i being the largest startpoint of an interval ending before i. The answer is $\mathrm{dp}[n+1]$.

Observation: we can find these k_i by sorting the intervals.

Running time: $\mathcal{O}(n^2)$, which is too slow.

Faster Solution: Use a segment tree to maintain the DP values.

Running time: $\mathcal{O}(n \log n)$, which is fast enough!

Easier Solution: Use a heap to maintain the DP values.

Fastest Solution: Use a deque to maintain the pareto front of $(i, \mathrm{dp[i]})$.

Running time: $\mathcal{O}(n)$ using bucket sort.

Pitfall: Use sufficiently large integers.

```
-    int sol = 2e9;
+    int sol = (1LL << 55);
```

```
 using namespace std;
+#define int long long
 typedef long long ll;
```

M: Museum Visit

Problem author: Tobias Roehr

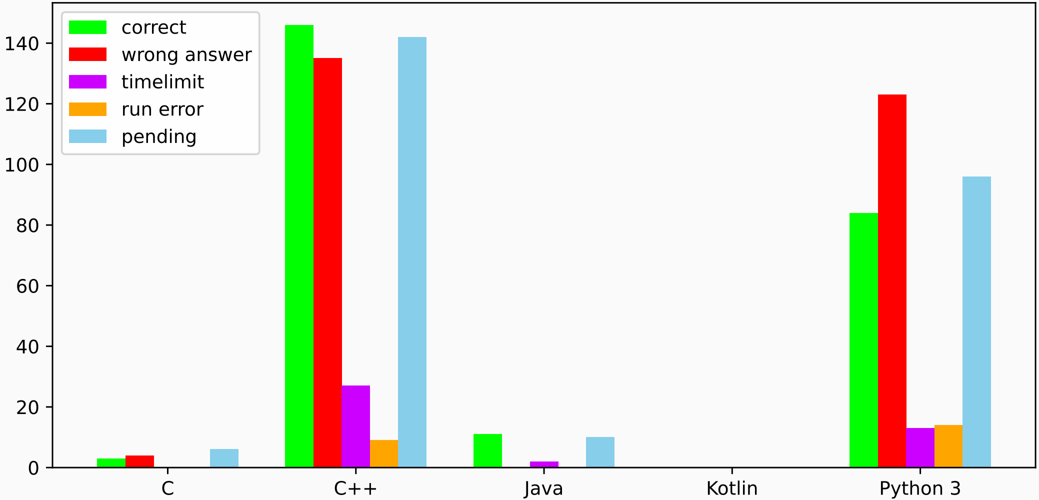
Pitfall: Use sufficiently large integers.

```
-    int sol = 2e9;
+    int sol = (1LL << 55);
```

```
 using namespace std;
+#define int long long
 typedef long long ll;
```

Statistics: 26 submissions, 5 accepted, 14 unknown

Language stats



Random facts

Jury work

- 1138 commits, of which 630 for the main contest (last year: 1061/564)

Random facts

Jury work

- 1138 commits, of which 630 for the main contest (last year: 1061/564)
- 1192 secret test cases (last year: 1358) ($\approx$ 91.7 per problem!)

Random facts

Jury work

- 1138 commits, of which 630 for the main contest (last year: 1061/564)
- 1192 secret test cases (last year: 1358) ($\approx$ 91.7 per problem!)
- The CI job for Horse Habitat costs 45 minutes, or ~€0.01 worth of heating Ragnar's apartment

Random facts

Jury work

- 1138 commits, of which 630 for the main contest (last year: 1061/564)
- 1192 secret test cases (last year: 1358) ($\approx$ 91.7 per problem!)
- The CI job for Horse Habitat costs 45 minutes, or ~€0.01 worth of heating Ragnar's apartment
- 273 jury + proofreader solutions (last year: 196)

Random facts

Jury work

- 1138 commits, of which 630 for the main contest (last year: 1061/564)
- 1192 secret test cases (last year: 1358) (≈ 91.7 per problem!)
- The CI job for Horse Habitat costs 45 minutes, or ~€0.01 worth of heating Ragnar's apartment
- 273 jury + proofreader solutions (last year: 196)
- The minimum[1] number of lines the jury needed to solve all problems is

$$3 + 10 + 7 + 16 + 3 + 23 + 3 + 16 + 7 + 1 + 2 + 13 + 11 = 115$$

  On average, 8.8 lines per problem (7.0 in BAPC 2023, 14.1 in preliminaries 2024)

[1] With PEP 8 compliant code golfing

Is it really PEP 8 compliant?

Yes, this submission for Jumbled Scoreboards is PEP 8 compliant!

```python
print("yneos"[any(sorted(t) != list(t) for t in zip(*(map(int, input().split()) for _ in range(int(input())))))::2])
```

Is it really PEP 8 compliant?

Yes, this submission for Jumbled Scoreboards is PEP 8 compliant!

```
print("yneos"[any(sorted(t) != list(t) for t in zip(*(map(int, input().split()) for _ in range(int(input()))))))::2])
```

And so is this submission for Grocery Greed. . .

```
A = input() and input().split()
a, b, c, d = [sum(int(x[-1]) % 5 == i for x in A) for i in range(1, 5)]
print(f'{sum(map(float, A)) - (a + 2 * b + 2 * (r := min(c, d)) + (c - r) // 2 + (d - r) // 3 * 2) / 100:.2f}')
```

Is it really PEP 8 compliant?

Yes, this submission for Jumbled Scoreboards is PEP 8 compliant!

```python
print("yneos"[any(sorted(t) != list(t) for t in zip(*(map(int, input().split()) for _ in range(int(input())))))::2])
```

And so is this submission for Grocery Greed. . .

```python
A = input() and input().split()
a, b, c, d = [sum(int(x[-1]) % 5 == i for x in A) for i in range(1, 5)]
print(f'{sum(map(float, A)) - (a + 2 * b + 2 * (r := min(c, d)) + (c - r) // 2 + (d - r) // 3 * 2) / 100:.2f}')
```

And this one for Karaoke Compression. . .

```python
b, a, r, m, f, e = (n := len(s := input() + '$#')), sorted(s[i:] for i in range(len(s))), range, min, next, enumerate
print(m(1 + b, m((o := f(j for j in r(n) if a[i - 1][j] != c[j])) + n - (o - 1) * s.count(c[:o]) for i, c in e(a))) - 2)
```

Is it really PEP 8 compliant?

Yes, this submission for Jumbled Scoreboards is PEP 8 compliant!

```
print("yneos"[any(sorted(t) != list(t) for t in zip(*(map(int, input().split()) for _ in range(int(input()))))))::2])
```

And so is this submission for Grocery Greed...

```
A = input() and input().split()
a, b, c, d = [sum(int(x[-1]) % 5 == i for x in A) for i in range(1, 5)]
print(f'{sum(map(float, A)) - (a + 2 * b + 2 * (r := min(c, d)) + (c - r) // 2 + (d - r) // 3 * 2) / 100:.2f}')
```

And this one for Karaoke Compression...

```
b, a, r, m, f, e = (n := len(s := input() + '$#')), sorted(s[i:] for i in range(len(s))), range, min, next, enumerate
print(m(1 + b, m((o := f(j for j in r(n) if a[i - 1][j] != c[j])) + n - (o - 1) * s.count(c[:o]) for i, c in e(a))) - 2)
```

But this one-liner for Extraterrestrial Exploration is *not*.

```
print('!',1,(n:=int(input())),[(u:=__import__('functools').reduce(lambda x,y:[x[0],sum(x)//2,x[1]][2*q(sum(x)//2)<v:][:2],'.'*20,(2,n-1,(q:=lambda x:int(input('? %i\n'%x)))and 0,0*(v:=q(1)+q(n))))[1]),u-1][q(u-1)+q(u)>v])
```

Thanks to:

The proofreaders

Arnoud van der Leer
Jaap Eldering
Jeroen Bransen (Java Hero 🎈)
Kevin Verbeek
Michael Vasseur
Mylène Martodihardjo
Pavel Kunyavskiy (Kotlin Hero 🎈)
Wendy Yi

The jury

Gijs Pennings
Jonas van der Schaaf
Jorke de Vlas
Lammert Westerdijk
Maarten Sijm
Mees de Vries
Mike de Vries
Ragnar Groot Koerkamp
Reinier Schmiermann
Thore Husfeldt
Tobias Roehr
Wietze Koops

Want to join the jury? Submit to the Call for Problems of BAPC 2025 at:

`https://jury.bapc.eu/`