Benelux Algorithm Programming Contest (BAPC) 2026

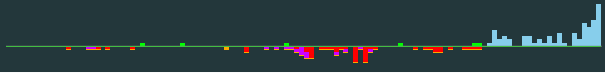
Solutions presentation

The BAPC 2026 Jury

November 2, 2026

A: Alternating Branch Cutting

Problem author: Yanxu Chen



Problem: Given that each leaf of a ternary of height n is green with probability $0.5 + p$. After Alice and Emmy have pruned the tree, what is the probability that the final leaf is green?

Small case: For $n = 1$, we need a majority green, which has probability

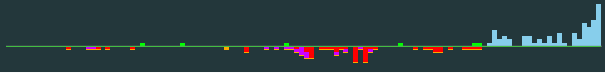
$$(0.5 + p)^3 + 3(0.5 + p)^2(0.5 - p) = 0.5 + p(1.5 - 2p^2).$$

Next case: For $n = 2$, each subtree of height 1 results in a green leaf with probability $0.5 + p_1$ for $p_1 = p(1.5 - 2p^2)$. We need a majority of these subtrees to result in a green leaf, which has probability $0.5 + p_2$ for $p_2 = p_1(1.5 - 2p_1^2)$.

General case: Define $p_0 = p$ and $p_{i+1} = p_i(1.5 - 2p_i^2)$. Then the answer is $0.5 + p_n$.

A: Alternating Branch Cutting

Problem author: Yanxu Chen



Pitfall: The input does not fit inside an extended precision floating-point value, due to the massive exponent. Need to manually keep track of the mantissa and the exponent and simply estimate $p_{i+1} = 1.5p_i$ when p_i is still very small.

Pitfall: Make sure negative values are handled correctly.

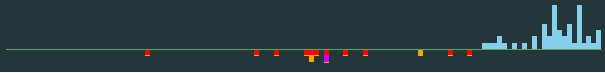
Pitfall: `decimal` in PyPy is slow. `p**3` is $30x$ slower than `p*p*p`. `decimal` in CPython is also 4 times faster...

Running time: Linear $\mathcal{O}(n)$. Also possible in constant time $\mathcal{O}(1)$ by exploiting the loose precision requirements, only explicitly calculating p_i when $10^{-20} < |p_i| < 0.5 - 10^{-10}$.

Statistics: 139 submissions, 6 accepted, 72 unknown

B: Balloons and Dubloons

Problem author: Marijn Adriaanse



Problem: Given a treasure map with fake markers, add a single marker to communicate the treasure.

Observation: The water does not matter, neither does the map. We can flatten the problem into $n + k$ land tiles in a row, with some marked.

Equivalent: Find a hash function mapping a set of $k + 1$ markers to the treasure, used by the matey. The hash should allow the captain to always add a working marker.

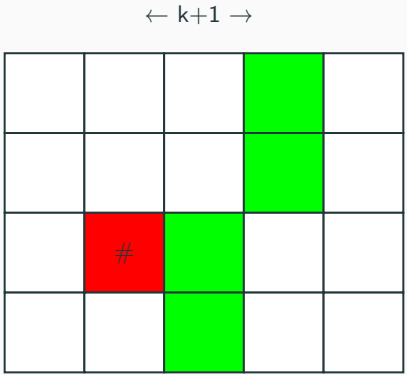
Solution: Take the sum of all marker indices, modulo $k + 1$, as the hash.

Pitfall: Do not recompute the hash for every marker location. This is too slow.

Pitfall: Taking the marker index from left to right does not always work! A fix is to index from right to left instead.

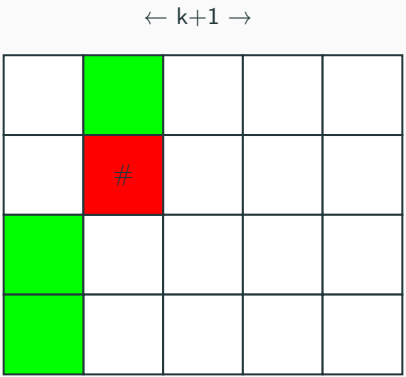
B: Balloons and Dubloons

Problem author: Marijn Adriaanse



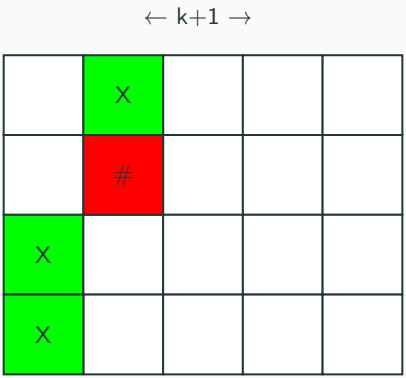
B: Balloons and Dubloons

Problem author: Marijn Adriaanse



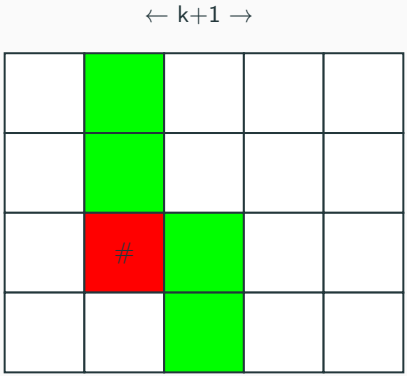
B: Balloons and Dubloons

Problem author: Marijn Adriaanse



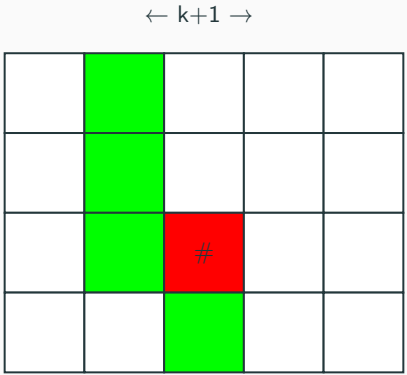
B: Balloons and Dubloons

Problem author: Marijn Adriaanse



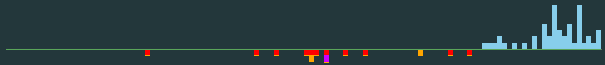
B: Balloons and Dubloons

Problem author: Marijn Adriaanse



B: Balloons and Dubloons

Problem author: Marijn Adriaanse



Cheese: Your hash does not need to be perfect. You can hardcode small cases, and randomly permute the maps on larger inputs to pass all tests.

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

Fun Fact: The bound $n > k^2$ can be lowered to $n \geq \Omega(k \log k)$, using the Lovász local lemma (the other LLL).

Statistics: 61 submissions, 0 accepted, 47 unknown

C: Circadian Rhythm

Problem author: Reinier Schmiermann

Problem: Solve two simultaneous preemptive scheduling problems, maximizing the overlapping running time.

Greedy 1: To check whether the scheduling problems are possible, we can use a greedy algorithm:

Simply always work on the task with the earliest deadline. This can be implemented by keeping the active tasks in a priority queue, adding new tasks once you reach their start time.

Greedy 2: To check for the maximum possible overlap, consider omitting the requirement that each task is completed. Then we can use the following greedy algorithm:

Simply only work on tasks if both problems have an active task, and then work on those with the earliest deadline. The implementation is very similar to Greedy 1.

Claim: If both scheduling problems are possible, then it is also possible with the maximum overlap found by Greedy 2.

C: Circadian Rhythm

Problem author: Reinier Schmiermann

Claim: If both scheduling problems are possible, then it is also possible with the maximum overlap found by Greedy 2.

Observation: Start with the schedules obtained by Greedy 2, and consider fixing the schedules of each problem separately. This can be modeled as a bipartite maximum matching problem, subdividing the available time and the required time of each task into individual milliseconds!

Proof of claim: Starting with the matching obtained by Greedy 2, the Ford-Fulkerson algorithm can turn this into a maximum matching. Since this algorithm never unmatches a vertex, we never lose any of the overlap!

Running time: It suffices to run Greedy 1 and Greedy 2, giving a total running time of $\mathcal{O}(n \log n + m \log m)$.

Update: After the contest, Greedy 2 was proven to be incorrect. Fixing it requires much more complicated techniques.

Extra challenge: The presented solution is non-constructive, but it is also possible to solve the problem constructively with the same asymptotic time complexity!

Statistics: 4 submissions, 0 accepted, 2 unknown

D: Discception

Problem author: Lammert Westerdijk



Problem: Given radii of n circles in array $r_1, r_2, \ldots, r_n$. Check if it is possible to place the circles on the plane so the centre point of each circle lies on the boundary of some other circle.

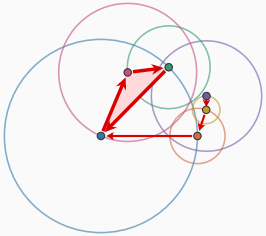
Observation 1: Assume solution exists. For each circle, set $p_i = j$, if the centre of i lies on the circle j. The edges (i, p_i) form a functional graph. This graph has a directed cycle.

Consequence: This directed cycle forms a polygon with edge lengths r_i.

Solution: If you can make a polygon out of a subset of the r_i, put the circle centres of these circles on the corners of the polygon.

The rest of the circles can be placed on one of the already placed circles.

Goal: Find a subset of r_i that can be the edges of a polygon, this is a necessary and sufficient condition.



D: Discception

Problem author: Lammert Westerdijk



Problem: Given radii of n circles in array $r_1, r_2, \ldots, r_n$. Check if it is possible to place the circles on the plane so the centre point of each circle lies on the boundary of some other circle.

Observation 2: Given edge lengths $w_1, w_2, \ldots, w_k$, these edges can form a (possibly degenerate) polygon iff:

$$2 \max(w_i) \leq \sum w_i \ (*)$$

Solution Cont. Sort array r_i: $r'_1 \leq r'_2 \leq \cdots \leq r'_n$

Assume r'_x is the longest edge. To maximize the RHS of $(*)$, take all edges $r'_1 \leq r'_2 \leq \cdots \leq r'_{x-1}$ in the polygon.

After sorting, a linear sweep through x while maintaining the prefix sum solves the problem.

Running time: $\mathcal{O}(n \log n)$ for sorting, rest is linear.

Fun fact: For all $n \geq 30$, simply output `possible`.

Statistics: 122 submissions, 36 accepted, 14 unknown

E: Endurtaka Mig

Problem author: Jeroen Op de Beek



Problem: Given string of length n, consisting of lower case characters, split it up into minimum number of substrings, all of which are not perfect powers.

A string s is a perfect power if it can be written as t^k (string t repeated k times), for $k > 1$.

Solution: First check if the answer is 1, by checking if s is a power.

Observation 1: If $s = t^k$, $|t|$ must be a divisor of n.

Observation 2: For $n \leq 3 \cdot 10^5$, n has at most 180 divisors.

Algorithm: Try all divisors d of n, take $t = s[0 : d]$ and check whether $t^k = s$, $\mathcal{O}(d(n)n)$ overall.

Observation 3: If $s = \texttt{aaa}\ldots\texttt{aaa}$, then the answer is n, because any substring with length greater than 1 is a perfect power.

Observation 4: If the answer is not 1 or n, the answer is 2!

E: Endurtaka Mig

Problem author: Jeroen Op de Beek



Observation 4: If the answer is not 1 or n, the answer is 2!

Proof: Assume $s = t^k$, for $k > 1$ (otherwise answer is 1).
Take the splitting $s[0] + s[1 : n]$. $s[0]$ is a single character, so not a perfect power
If $s[1 : n]$ is a perfect power, then it can be written as u^m for some string u and $m \geq 2$.
For the string $s[1 : n]$, we know $s[i] = s[i + |t|]$ and $s[i] = s[i + |u|]$ for all i, because s is a power of t, and $s[1 : n]$ a power of u.

Gcd calculations: $\gcd(n, n-1) = 1$, and $|t|$ and $|u|$ are divisors of n and $n-1$, so $\gcd(|t|, |u|) = 1$.

Periodicity: Because $|t| + |u| \leq n-2$, these equality constraints force s to have all equal characters.

Details: Standing at position i, making hops of $\pm|t|$ and $\pm|u|$, prove there always exists hop sequence to position 0.

Conclusion: If the answer is not 1 or n, then $s[1 : n]$ is not a perfect power, so $s[0] + s[1 : n]$ is a valid split.

Running time: $\mathcal{O}(nd(n))$. Results in $\approx 6 \cdot 10^7$ simple operations.

Bonus: Can you find an $\mathcal{O}(n)$ solution?

Statistics: 92 submissions, 26 accepted, 14 unknown

F: Flippin' Cards

Problem author: Mike de Vries

Problem: You must orient n cards, each card is upside down with probability p independent of the other cards (p is drawn uniformly from $[0, 1]$). You go from card 1 to card n. If card i is right side up, then it takes time x. If card i is upside down, then either you flip card i in time y or you flip all cards $i, i+1, \ldots, n$ in time z. Determine the expected time needed to orient all cards using an optimal strategy.

Observation: Suppose you have seen c cards. The only thing that impacts the conditional probability on p (and thus your strategy) is the number of cards that are upside down u.

Solution: Denote with $f(c, u)$ the expected time to orient the remaining $n - c$ cards if you have seen c cards and u of them were upside down. If $q(c, u)$ denotes the probability that the next card is upside down, then you get:

$$f(c,u) = (1 - q(c,u)) \cdot (f(c+1,u) + x) + \\ q(c,u) \cdot \min(f(c+1,u+1) + y, f(c+1,c-u) + z)$$

The second alternative describes performing move 3: all remaining cards flip, which we mimic by saying that exactly the previous $c - u$ correct cards were "upside down".

Algorithm: Calculate $f(c, u)$ using dynamic programming in $\mathcal{O}(n^2)$ and we are done.

F: Flippin' Cards

Problem author: Mike de Vries

Are we done? In this algorithm, we need to know $q(c, u)$ (= the probability that the next card is upside down if u cards out of the seen c cards are upside down). How to calculate it?

Calculation: For fixed p, the probability of u cards out of c upside down is $\binom{c}{u} \cdot p^u \cdot (1-p)^{c-u}$.
Hence for random p, the probability becomes: $\int_0^1 \binom{c}{u} \cdot x^u \cdot (1-x)^{c-u} dx$
For fixed p, the probability that the next card is upside down is $\binom{c}{u} \cdot p^u \cdot (1-p)^{c-u} \cdot p$.
Hence, we find:

$$q(c,u) = \frac{\int_0^1 \binom{c}{u} \cdot x^{u+1} \cdot (1-x)^{c-u} dx}{\int_0^1 \binom{c}{u} \cdot x^u \cdot (1-x)^{c-u} dx} = \frac{\int_0^1 x^{u+1} \cdot (1-x)^{c-u} dx}{\int_0^1 x^u \cdot (1-x)^{c-u} dx}$$

How to evaluate this?

1. **Numeric:** We can apply numeric integration (for example, Simpson's Rule) to calculate this for all $q(c, u)$: doable, but can be tricky to get past time limit.

2. **Heavy Math:** Note that $\int_0^1 x^a(1-x)^b dx$ is the beta-function: it evaluates to

$$\int_0^1 x^a(1-x)^b dx = \frac{a! \cdot b!}{(a+b+1)!}$$

This formula can be found by trying some small a and b by hand (and can be proven by AC).
Additionally, we can prove it with induction and partial integration.

Plug in and we get:

$$q(c,u) = \frac{\int_0^1 x^{u+1} \cdot (1-x)^{c-u} dx}{\int_0^1 x^u \cdot (1-x)^{c-u} dx} = \frac{(u+1)! \cdot (c-u)! \cdot (c+1)!}{u! \cdot (c-u)! \cdot (c+2)!} = \frac{u+1}{c+2}$$

3. **Beautiful Math:** Equivalent formalization: for card i, draw uniform random number x_i from $[0, 1]$, card i is upside down if $x_i < p$. Recall that p is also drawn uniformly from $[0, 1]$.

Observation: We have $c + 1$ numbers $x_1, \ldots, x_c, p$ drawn uniformly from $[0, 1]$ and we are only interested in their order (or specifically, the location of p in their order): the order $x_1, \ldots, x_c, p$ is uniformly random over all permutations.

Consequence: If we draw c cards and u are upside down, then this means that u cards x_i are put before p and $c - u$ cards are put after c. For the conditional probability $q(c, u)$ that card $c + 1$ is upside down, we must consider all ways to put card x_{c+1} in the order of $x_1, x_2, \ldots, x_c, p$ and all options are equally likely. There are $u + 1$ locations before p (in which card $c + 1$ is upside down) and $c - u + 1$ locations after p (in which card $c + 1$ is right side up), so $q(c, u) = \frac{u+1}{c+2}$.

Fun fact: Note that this technique immediately proves the formula for the beta-function (for non-negative integers a, b).

F: Flippin' Cards

Problem author: Mike de Vries

Wrap up: We know that $q(c,u) = \frac{u+1}{c+2}$. This gives the dynamic program:

$$f(c,u) = \frac{c-u+1}{c+2} \cdot (f(c+1,u)+x) + \frac{u+1}{c+2} \cdot \min(f(c+1,u+1)+y, f(c+1,c-u)+z)$$

Calculate it from $c = n$ to $c = 0$ for all $0 \le u \le c$ with base case $f(n,u) = 0$ for all u. The answer is $f(0,0)$.

Running time: $\mathcal{O}(n^2)$

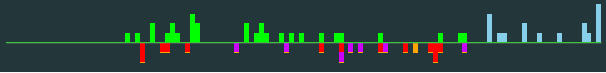
Fun fact: $\mathcal{O}(n\sqrt{-n\log\varepsilon})$ is possible

Fun fact 2: For $c := \frac{z-x}{n(y-x)} < 1$, the answer is asymptotically $\left(\frac{1}{2} - \frac{1}{4}(1-c)^2\right) \cdot n + \mathcal{O}(\sqrt{n})$, which can be obtained by only flipping all cards after $\sqrt{n}$ moves if at least $\frac{c+1}{2}\sqrt{n}$ cards were upside down.

Statistics: 0 submissions, 0 accepted

G: Graph Theft Auto

Problem author: Lammert Westerdijk



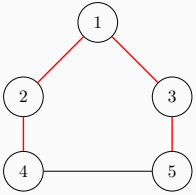
Problem: On an undirected unweighted graph, some gangsters walk from their hideout at unknown vertex to vertex 1 using the shortest path. You can place cameras at vertices that detect if the gangsters walk past this vertex and the number of edges they have traversed so far. Determine the minimum number of required cameras to spot the gangsters and be able to determine their hideout.

Step 1: Determine the shortest paths. We can calculate the shortest path from any starting vertex to all other vertices in $\mathcal{O}(n+m)$ running time using BFS.

Run BFS for all starting vertices is too slow ($\mathcal{O}(n^2+mn)$).

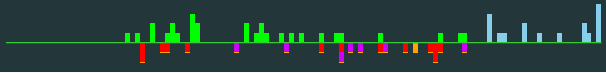
We can run BFS from vertex 1: $\mathcal{O}(n+m)$ is fast enough.

The gangsters always walk upward to the root of the BFS tree (forget all other edges):



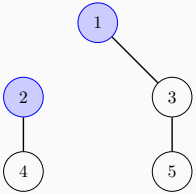
G: Graph Theft Auto

Problem author: Lammert Westerdijk



Step 2: Place cameras optimally.

Observation: If different hideouts are not distinguished by a camera, then they will not be distinguished higher in tree as well. We can cut edges from cameras to parent and treat subtrees separately:



Condition: We must distinguish all hideouts: subtrees are all paths (2 children not distinguishable).

Solution: All leaves need their own path. We can create these paths by cutting off as high as possible: the answer is the number of leaves in BFS-tree!

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

Statistics: 66 submissions, 29 accepted, 16 unknown

H: Historical Hierarchy

Problem author: Reinier Schmiermann



Problem: There is a hidden permutation $p_1, \ldots, p_n$. In one step, you are allowed to remove an element from position i, and insert it at position j instead. After each operation you get told the length of a Longest Increasing Subsequence (LIS). Sort the permutation in at most $4n^2$ operations.

Naive Idea: Keep trying random pairs i j, and undoing the moves which do not increase the LIS, until LIS $= n$. You can undo an i j operation with the operation j i. This takes too many queries.

Solution? Based on the Naive Solution, try to gradually improve the LIS to n, but use fewer queries than random operations.

Number the elements from 1 to n from left to right (although this numbering may differ from the hidden permutation $p_1, \ldots, p_n$). Then store the array a_i which is the current permutation in your own numbering system.

Algorithm: For all x from 1 to n, find the element with $a_i = x$, and try deleting it and placing it everywhere (using operations i 1, 1 2, 2 3, $\ldots$, $n-1$ n)

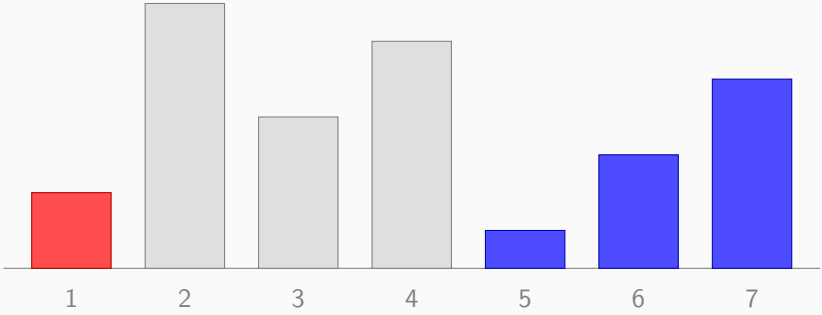
Then choose to place $a_i = x$ at position which gave the maximum LIS.

Operations: This takes $n(n+1) < 4n^2$ operations, which easily fits within the limits.

But why does this algorithm sort the permutation in the end?

H: Historical Hierarchy

Problem author: Reinier Schmiermann

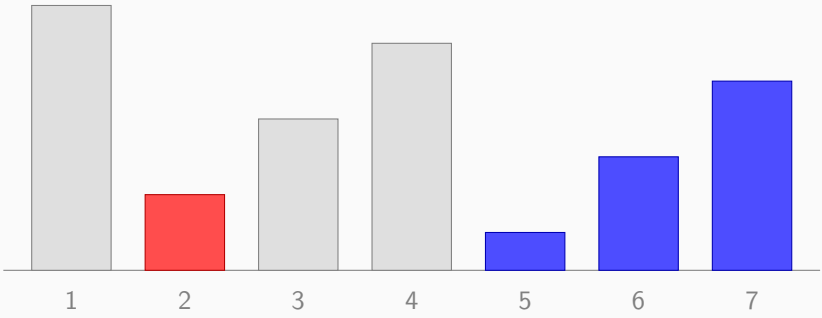


LIS: 3

1 1

H: Historical Hierarchy

Problem author: Reinier Schmiermann

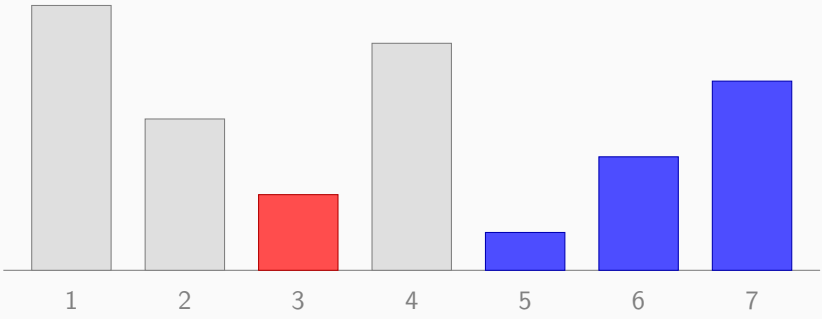


LIS: 3 3

1 2

H: Historical Hierarchy

Problem author: Reinier Schmiermann

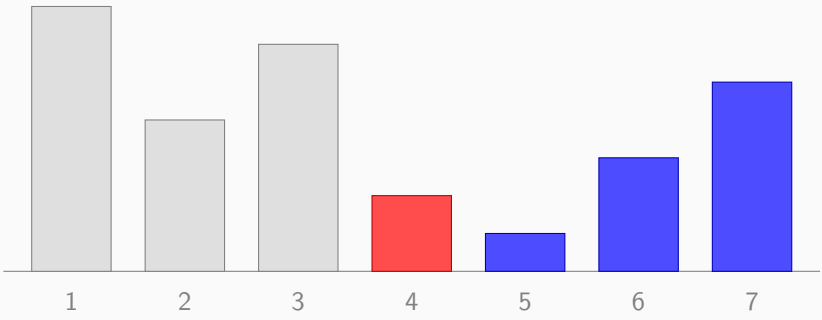


LIS: 3 3 3

2 3

H: Historical Hierarchy

Problem author: Reinier Schmiermann

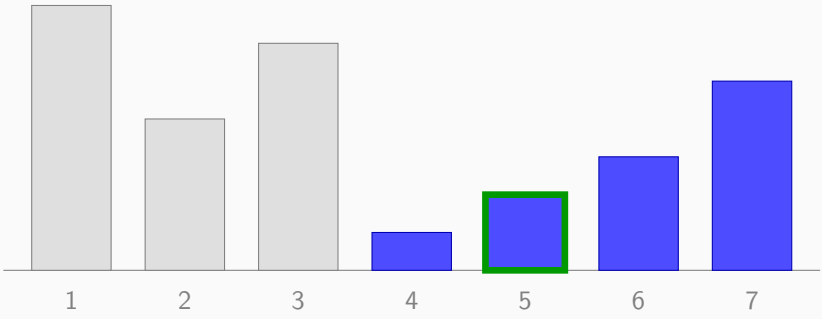


LIS: 3 3 3 3

3 4

H: Historical Hierarchy

Problem author: Reinier Schmiermann

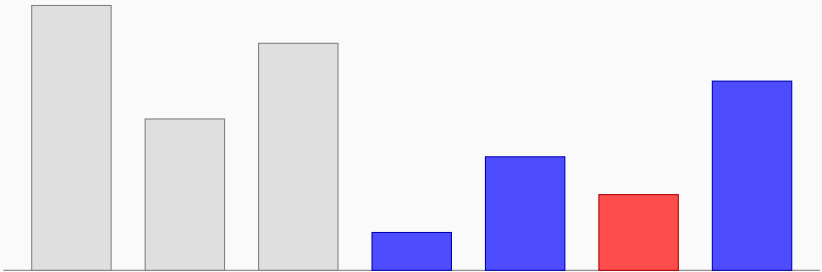


LIS: 3 3 3 3 **4**

4 5

H: Historical Hierarchy

Problem author: Reinier Schmiermann

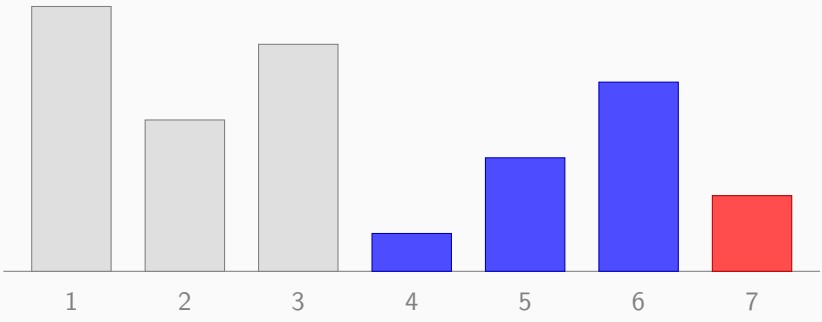


LIS: 3 3 3 3 4 3

5 6

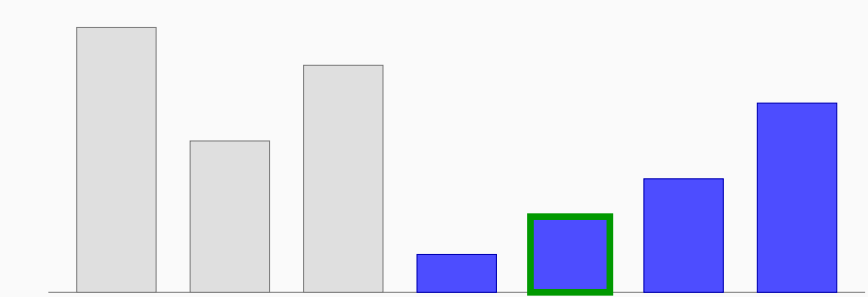
H: Historical Hierarchy

Problem author: Reinier Schmiermann



H: Historical Hierarchy

Problem author: Reinier Schmiermann



7 5

H: Historical Hierarchy

Problem author: Reinier Schmiermann



Algorithm: For all x from 1 to n, find the element with $a_i = x$, and try deleting it and placing it everywhere. Choose to place $a_i = x$ at position of max LIS.

But why does this algorithm sort the permutation in the end?

Issue: Multiple increasing subsequences that are tied for longest make things confusing.

Proof: Prove that after maximizing the LIS for the element $x = a_i$ the elements that we labeled $1, 2, \ldots, x$ are all part of all LISes $(*)$.

Induction Step: Assume $(*)$ is true for $x - 1$.

Option 1: x is already part of all LISes, then wherever it moves, the LIS can only stay equal or go down. In this case we should keep x on its original position, before the operations. Then $(*)$ holds for this x as well.

Option 2: x is not part of some LIS. Then in some operation it gets inserted at the right spot inside the gaps of this LIS, So there is a position which makes the length of LIS increment. There could be multiple previous LIS which now increased in length by 1. But by assumption, elements 1 to $x - 1$ are in all LISes, so also the ones that grew to length+1, so $(*)$ still holds after this step.

H: Historical Hierarchy

Problem author: Reinier Schmiermann



Fun Fact: It is possible to solve the problem in $n \log_2 n + \mathcal{O}(n)$ queries with high probability. Ask Mike for details.

Statistics: 48 submissions, 9 accepted, 28 unknown

I: Irritating Interruption II

Problem author: Tobias Röhr



Problem: Repeatedly assign mathematical operations $(+, \times)$ to armies to maximize their combined strength.

Observation: The effect of assigning additions to armies is determined by future multiplications.

Observation: Additions are best assigned such that the largest addition continues to all the largest multiplications. *(proof left as an exercise to the reader)*

Greedy solution: Send the strongest army through the best gate.

Why: All the biggest additions end up with the largest multiplications.

Result: This works!

Running time: $\mathcal{O}(nk \log k)$.

Statistics: 85 submissions, 48 accepted, 2 unknown

J: Juxtaposort

Problem author: Lammert Westerdijk



Problem: Sort a list of values, if unambiguously possible, using the results of all $a_i - 1 < a_j$ comparisons.

Observation: Comparing numbers a and b both ways results in the cases $a \leq b - 1$ ($a \ll b$), $b - 1 < a < b + 1$ ($a \approx b$), $b + 1 \leq a_i$ ($a \gg b$)

First thought: Toposort on $a \ll b$.

Result: This fails if $a \approx b$, $b \approx c$, $a \ll c$ (sample 3).

Observation: If $a \approx b$, the list can only be sorted if there exists such a c to differentiate between them.

Solution: For each pair $a \approx b$, look for a c that determines their order. Then toposort.

Result: $\mathcal{O}(n^3)$ is probably too slow (note: $\mathcal{O}(n^3/\mathrm{bitset})$ is doable).

J: Juxtaposort

Problem author: Lammert Westerdijk



Observation: Comparing with one number divides the other numbers into 3 sets.

Solution: Keep a sorted list of sets of unsorted values, and compare with each number to partition these sets.

Result: This works.

Running time: $\mathcal{O}(n^2)$.

...but it is needlessly complicated!

J: Juxtaposort

Problem author: Lammert Westerdijk



Simple solution: Give each value a_i a score s_i, then sort and check for ambiguity.

Implementation: For each $a_i - 1 < a_j$, do `s[i]--` and `s[j]++`.

Result: This works.

Why: If $a_i < a_j$, s_j will get at least as many increments as s_i, and s_i will get at least as many decrements as s_j.

Running time: $\mathcal{O}(n^2)$.

Statistics: 31 submissions, 5 accepted, 18 unknown

K: Kaktus Kourse

Problem author: Leon van der Waal

Problem: Find the length of the k-th longest simple path in a kaktus graph.

Naive solution: Loop over all starting points, and brute-force all possible paths using DFS.

Verdict: **TLE**, but what is the complexity?



Observation: Fix the first and last node of a simple path. There are 2^c possible simple paths between these nodes, where c is the number of cycles inbetween these nodes.

K: Kaktus Kourse

Problem author: Leon van der Waal



Observation: Fix the first and last node of a simple path. There are 2^c possible simple paths between these nodes, where c is the number of cycles inbetween these nodes.

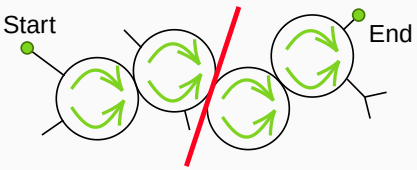
Observation: c is at most $\left\lfloor \frac{m}{3} \right\rfloor$, since each cycle has at least 3 edges.

Observation: The worst case has $\left\lfloor \frac{m}{3} \right\rfloor$ cycles in a line.

Observation: By geometric sums, there can only be $\mathcal{O}\left(2^{\frac{m}{3}}\right)$ simple paths.

Running time: Finding all possible simple paths is too slow, we need to do something more efficient.

Better solution: Do binary search on the answer. This way, we have to count the number of simple paths of length $\leq B$, when we binary search on B.
For all start and end nodes, let there be k cycles inbetween the start and end node.



Separate the first $\left\lceil \frac{k}{2} \right\rceil$ and last $\left\lfloor \frac{k}{2} \right\rfloor$ cycles.

For both halves, precompute and sort all possible path lengths in $\mathcal{O}\left(2^{\frac{k}{2}}\right)$ time.

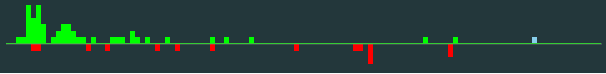
For a binary search bound B, we can count the number of simple paths in $\mathcal{O}\left(2^{\frac{k}{2}}\right)$, using two pointers.

Running time: By geometric sums, the overall complexity is hence $\mathcal{O}\left(2^{\frac{m}{6}} \log(nA)\right)$, with $A = 10^9$.

Statistics: 4 submissions, 0 accepted

L: Letter Lottery

Problem author: Maarten Sijm



Problem: In a raffle, there are n codes, one is uniformly selected at random. A code consists of lowercase letters, and once the code is selected, a random letter is taken from this code. What is the lowercase letter with the highest probability? For that letter, output the probability.

Observation: The probability that letter j is chosen from code i is

$$\frac{1}{n} \cdot \frac{1}{|b_i|}.$$

Solution: Add up the probabilities for each lowercase letter, and output the maximum probability out of the letters.

Pitfall: Be sure to use 64-bit floating point numbers. 32-bit floating point numbers are not precise enough.

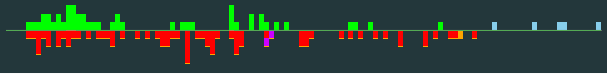
Fun fact: Originally, jury wanted the output to be the letter with max probability. There can be very near ties, so this version needs big integer fractions.

Running time: $\mathcal{O}\left(\sum |b_i|\right)$.

Statistics: 66 submissions, 48 accepted, 1 unknown

M: Memory Management

Problem author: Jeroen Op de Beek, Wietze Koops



Problem: For a generalized FizzBuzz sequence, with Fizz at multiples of a and Buzz at multiples of b, what is the largest k for which the answer string still fits in a memory of n bytes.

Solution: Simulate Fizz Buzz, and stop once the memory limit is reached.

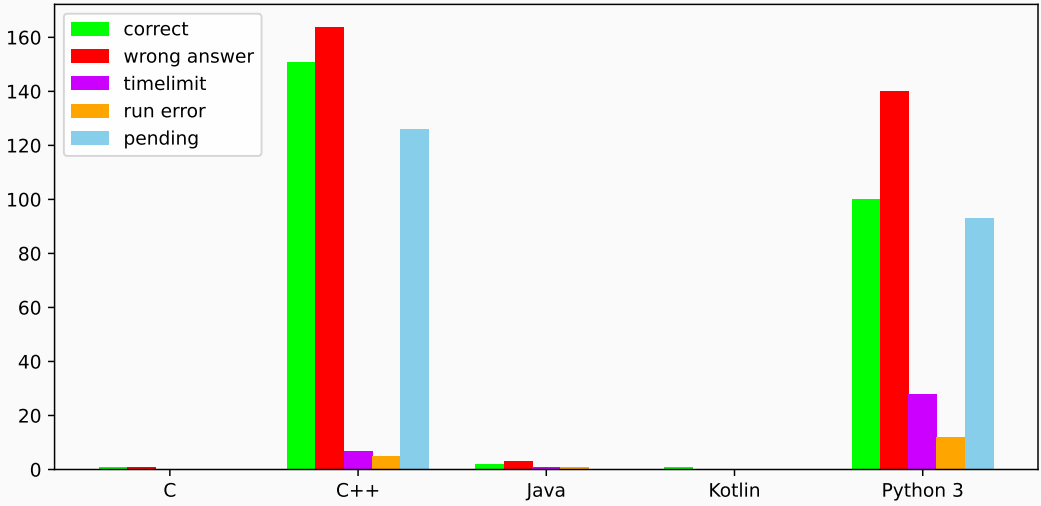
Running time: $\mathcal{O}(n)$.

Pitfall: When constructing the string, make sure to not run into a runtime of $O(n^2)$, due to slow concatenations.

Bonus: The problem can be solved in $O(\log n)$ with binary search, but this was not needed.

Statistics: 124 submissions, 47 accepted, 5 unknown

Language stats



Random facts

Jury work

- 825 commits, of which 449 for the main contest (last year: 945/525)
- 1245 secret test cases (last year: 1035) (≈ 95.8 per problem!)
- 290 jury + proofreader solutions (last year: 293)
- The minimum[1] number of lines the jury needed to solve all problems is

$$4 + 20 + 41 + 2 + 14 + 3 + 19 + 8 + 2 + 3 + 48 + 2 + 4 = 170$$

  On average, 13.1 lines per problem ($4\frac{1}{3}$ in BAPC 2025, 8.7 in preliminaries 2026)

[1] With limited code golfing

Thanks to:

The proofreaders

Arnoud van der Leer
Bartjan Henkemans
Hasso Plattner Institute team "Linked Liszt" 🎈
(Jakob Fanselow, Julian Windheuser, Konstantin Groth)
Hasso Plattner Institute team "Vikunjaaa" 🎈
(Helena Ullrich, Peter Zackrisson, Simon Bachran)
Jaap Eldering
Nicky Gerritsen 💥
Pavel Kunyavskiy Kotlin Hero 🎈
Thomas Verwoerd Kotlin Hero 🎈

The jury

Jeroen Op de Beek
Jippe Hoogeveen
Jonas van der Schaaf
Lammert Westerdijk
Leon van der Waal
Maarten Sijm
Marijn Adriaanse
Mike de Vries
Ragnar Groot Koerkamp
Reinier Schmiermann
Tobias Röhr
Wietze Koops
Yanxu Chen

Want to join the jury? Submit to the Call for Problems of BAPC 2027 at:

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