ICFP Contest 2024 Report

JEROEN BRANSEN, —, Netherlands
MATÚŠ TEJIŠČÁK, Chordify, Netherlands

1 INTRODUCTION

The ICFP Contest[1] (ICFPC) is a yearly programming contest held by the ACM SIGPLAN International Conference on Functional Programming[2] since 1998. This report describes the 2024 edition[3] of the contest, which was organized by a group of functional programming enthusiasts from the Netherlands. The report is written for contestants as well as future organizers, and contains many spoilers, so in case you have not seen the tasks yet and would like to approach it, we suggest you look at the task page[4] on the website instead.

The contest always takes 72 hours, and has some scoring that decides which team wins. After the first 24 hours, called the lightning round, a winner is also picked. Finally, after the contest the organizers award a special jury prize to a team that they feel deserve a prize despite not being at the top of the scoreboard. The prizes are not of monetary value, but instead the organizers make certain statements about the programming languages that the winning teams used, see Section 8 Results.

1.1 Timeline

The contest was held at the end of June, but the first brainstorm session dates back to more than a year before that. Some of us had a lot of fun replaying the 2006 ICFPC[5], and thought about a potential followup. In the fall of 2023 it was decided to that we would get to organize the contest, after which some ideas started to come to mind. In January the team was formed and we started writing down more concrete design criteria, which led to concretizing the rough ideas.

In March, we made decisions on the overall shape of the contest (a functional communication language with a set of relatively independent subtasks) as well as which subtasks to work out. While only two people did the actual programming for the whole contest, we comfortably made it on time by committing to decisions early in the year. On various occasions there were other nice ideas for subtasks, but looking back it was a good decision to stick to the ideas we initially committed to.

1.2 Design criteria

Before discussing a concrete problem idea, we settled on some more general criteria for making the contest a success. These were:

Fun for participants: there is no monetary prize, the only prizes are statements made by the jury, so the fun of working for 72 hours on the problem is the main reward for teams.

Connection with functional programming: many of the contestants are from outside the functional programming community, and we must make sure that the contest can be a fun experience for these teams as well. However, the ICFP is really about functional

[1] https://www.icfpconference.org/contest.html
[2] https://www.icfpconference.org/
[3] https://icfpcontest2024.github.io/
[4] https://icfpcontest2024.github.io/task.html
[5] http://www.boundvariable.org/

Authors' addresses: Jeroen Bransen, —, Netherlands; Matúš Tejiščák, Chordify, Netherlands.

programming, so we felt that there should be a clear functional programming component to the task.

Challenging: 72 hours is a lot of time, and teams from all over the world compete, so we must make sure that the teams do not "finish" everything before the end of the contest. The usual trick is by making it an optimization kind of problem which is infeasible to solve perfectly, so that score improvements can be made until the end the contest.

Low entry barrier: we should expect a lot of teams that are new to this contest as well, so we should aim for a problem where all teams can score points within a few hours.

Live scoreboard: for participants, this makes it much more fun, so we should make sure that we have a live scoreboard that keeps teams motivated to improve their scores even more.

Manageable server capacity: we should not underestimate the amount of submissions that come in, so problems that require significant computing power on our end might lead to running over the budget. The solutions must be hard to find but easy to verify.

Permanent artifact for the future: while the main goal should be organizing a fun contest, we actually enjoyed most of the older contests much later by rerunning them locally. Therefore, we should make sure we don't set things up only for the 72 hours of the contest, but make it so that in 10 years' time, people can still work on the task.

2 STORYLINE

Next to an interesting problem, we also wanted to have a storyline that would tie the subtasks together. Instead of coming up with a completely new story, we decided to take the story lines of some past ICFPCs which we enjoyed (2006, 2007, 2020) and combine these. This led to the following introduction to the task:

> As many will remember, in 2006 we discovered[6] documents of an old society, called the Cult of the Bound Variable. With the valuable help from the ICFP community, the Cult's computing device (Universal Machine) was brought back to life, and much interesting information could be recovered. In recent observations from the Pegovka observatory[7], we discovered something stunning: the Cult of the Bound Variable still exists, and has migrated their civilization to space! It is suspected that Endo[8] helped them move, after he escaped from the Earth again.
>
> After a couple of months of research, we have been able to decipher most of the received messages, and set up a communication channel. People of the Cult of the Bound variable now use *Interstellar Communication Functional Programs* (ICFP) to communicate. Our findings about ICFP expressions can be found on this page.

A communication channel was set up to communicate with the School of the Bound Variable, in which contestants are following courses (which we call subtasks here) and have to pass tests (problems). Since all communication happens with these so-called ICFP expressions, contestants first had to implement a part of this language in order to encode and decode the messages to read the real tasks. On the other hand, we designed the contest so that implementing just the encoder/decoder for string literals was sufficient for teams to get started, interact with the system, and solve many problems.

[6] http://www.boundvariable.org/task.shtml
[7] https://icfpcontest2020.github.io/
[8] https://save-endo.cs.uu.nl/

3 ICFP LANGUAGE

The ICFP language that we designed is a pure non-strict dynamically typed functional programming language, with Boolean, integer and string values. It is based on the lambda calculus with a call-by-name evaluation strategy. One particular property of the language is that it, on purpose, is not very human-readable; even string and integer constants are encoded in a non-standard way to make sure that contestants needed to write their own parser and printer for this language. Furthermore, we wanted to make it a functional language to put the functional programming in, small enough to make a somewhat concise task description, but rich enough to allow interesting expressions.

The full language description[9] can still be found on the website, we present a short summary here. An expression consists of a space-separated list of tokens, where the first character of each token indicates its type. Integers are base-94 encoded, strings use a character mapping (see Strings section), and for binary and ternary operators Polish prefix notation is used. For example, the expression

```
B$ B$ L# L$ v# B. SB%,,/ S}Q/2,$_ IK
```

could be shown in a more human-readable form as

$$((\lambda x.\ \lambda y.\ x)\ (\text{“Hello”}\ .\ \text{“World”}))\ 42,$$

which would evaluate to “Hello World!” (the `.` operator stands for string concatenation).

The built-in operators of the language are a relatively standard set of operations on booleans, integers and strings, as well as an **if** ternary operator.

3.1 Strings

Because we wanted to make sure that contestants needed to write custom code to read the tasks of the contest, all strings were encoded. In earlier versions of the language, the string encoding was “interpret this ICFP token as a number, then reinterpret its numeric value in base-128, and extract the ASCII string”. However, we decided it was not a good idea to require bigints for pretty much the most elementary task in the contest. We wanted that even less experienced teams with perhaps languages that do not support bigints can have fun, even if they can’t get in the top 10.

Then we changed the definition of strings to simple substitution, which led to another tricky decision: we wanted to be able to use spaces and newlines in the strings, but we could not use those characters in the encoding since we decided that tokens were space-separated. Therefore, we had only 94 available characters to encode 96 characters, so we had to remove two of them. We changed the available character set as we went, based on the needs of the subtasks (e.g. `3d` required `@` etc.). This was a bit tricky because that led us to removing `{` and `}` quite late in the preparations, which caused issues during the contest, as Haskell error messages tend to contain these characters, and they made our error message encoding routines panic.

3.2 Evaluation

We had several independent implementations. The two major ones were Haskell (small-step reduction) and Rust (big-step evaluation). Luckily, they did agree on the number of beta reductions, which we chose as the measure of running time. Later, we reimplemented a new big-step evaluator in Haskell for the actual contest.

The call-by-name evaluation was chosen because we felt that it is most easily explained in terms of beta reduction steps. However, it was easy to add the lazy and strict application operators as well so we did. We hoped that they would let people write more interesting programs but that turned out not to be the case; nobody seemed to really use them in the end.

[9] https://icfpcontest2024.github.io/icfp.html

After the contest some teams mentioned that they struggled to understand the binary application and how to deal with capture-avoiding substitutions, and felt the problem could have used a more extensive explanation. While we probably could have explained things better, this was also partially on purpose; while we are aware that many competing teams are from outside of the functional programming community, we wanted a functional programming component in the contest, so this is where functional programmers had a head start. And we did use a standard evaluation strategy, so there are many other sources explaining this. Alternatively, as some teams did, the expressions could also be translated to and from an existing functional language.

4 SUBTASKS

To earn points in the contest, there were 5 different subtasks for which points were assigned. At the beginning of the contest all of these were hidden, and as soon as the first team unlocked a subtask, a new column would be added to the scoreboard.

4.1 Hello

The `hello` subtask is not a real task, but instead a small set of different actions for which points were given away. We added this to make sure teams could quickly be on the scoreboard, and to teach them how interaction with the server works.

The following four actions were rewarded.

Hello1. Sending the ICFP-encoded version of `get index`, which could be copy & pasted from the task page. This would return the following welcome text.

> Hello and welcome to the School of the Bound Variable!
> Before taking a course, we suggest that you have a look around. You're now looking at the [index]. To practice your communication skills, you can use our [echo] service. Furthermore, to know how you and other students are doing, you can look at the scoreboard.
> After looking around, you may be admitted to your first courses, so make sure to check this page from time to time. In the meantime, if you want to practice more advanced communication skills, you may also take our [language_test].

Because the initial command was `get index` (but looks like `S'%4}).$%8` in ICFP), this page hints at the fact you now need to construct a string representing `get echo`, `get scoreboard` or `get language_test`.

Hello2. Sending `get scoreboard`, which is hinted at on the index page, was enough to give you the points. However, to get this far teams needed to implement string encoding and decoding.

Hello3. Sending `get echo` returns:

> The School of the Bound Variable provides a special echo service for you. If you send an ICFP expression evaluating to:
> `echo <some text>` it will respond with `<some text>`.
> Hint: you can use this to validate the Macroware Insight evaluator has the expected behavior. Of course the usual limitations still apply.

The points for `hello3` were assigned when correctly using the echo service, e.g. sending `echo x`. The last line was an important hint, as it can be valuable to check, when working on the more complex subtasks, how the server implementation behaves. For example, you could send something like **if** $((-1)/2) = 0$ **then** "echo yes" **else** "echo no", encoded as a suitable ICFP expression, to check how integer division rounds.

Hello4. As a reward for successfully evaluating the expression that you get with `get language_test`, the last points for the `hello` subtask could be scored. The expression is a big if-statement checking that most language constructs work as expected, and then evaluates to "Self-check OK, send 'solve language_test 4w3s0m3' to claim points for it" once they are all correct.

4.2 Spaceship

To make the contest beginner friendly, we wanted to include a problem that could be solved without too much fiddling around with ICFP, and took inspiration from ICFPC 2020[10] to do something with space. Initially the idea was more on something geometric, but we wanted to avoid non-whole numbers, and it's not easy to come up with a (not completely contrived) geometric problem with integral coordinates.

Originally, we came up with an idea based on chess, which we named the Travelling Salesqueen. The Salesqueen moves like the chess queen and has a list of places to visit. The teams have to come up with the shortest move sequence to visit them. Eventually, we came up with the idea of acceleration, which extended travelling salesqueen to a spaceship. Still with integral coordinates, but much more interesting behaviour, quite like in the 2020 contest.

So, the spaceship problem as presented to the contestants during the contest is as follows. The spaceship is a chess piece which has a speed v_x and v_y (both integers, initially 0). In each move, v_x and v_y can be increased or decreased by at most one, and then the spaceship moves v_x steps horizontally and v_y steps vertically. In each problem a set of target locations is given, and the task is to find a sequence of moves such that the spaceship visits all of the given locations. The score is the number of moves; the lower the better.

4.3 Lambdaman

Another reference that we wanted to make was to ICFPC 2014[11], which revolved around lambdaman. As we also wanted to do something with code golfing and our own ICFP language, we decided to base the score for this problem on the length of the submission, not on the number of moves. This also added to the "element of surprise": it might take some time for teams to realize that this problem is not actually on optimizing the number of moves.

In this problem, contestants are given a rectangle grid containing walls, pills and one starting position of the lambdaman, and must submit a sequence of moves for lambdaman such that all squares with pills are visited. The special sauce in this problem is the fact that 1. lambdaman may run into the walls; 2. we score the length of the generating program, not the number of steps.

Hence even though this problem looks much like spaceship on the surface (finding paths in 2D space), good solutions look and feel completely differently. What we aimed for is that contestants would not realize this immediately, but instead be surprised by the scores of some other teams which seem impossible at first glance. To help teams a little bit, we included `lambdaman6`:

```
B. SF B$ B$ L" B$ L" B$ L# B$ v" B$ v# v# L# B$ v" B$
v# v# L$ L# ? B= v# I" v" B. v" B$ v$ B- v# I" Sl I#,
```

While this expression is only 107 characters long, it generates a string of length 200: a lambdaman followed by 199 pills. The obvious solution to this problem would be 199 times `R`, but the best submitted solution to this problem is only 73 characters long:

```
B. S3/,6%},!-"$!-!.[} B$ L# B$ v# B$ v# B$ v# SLLLLLLLL L$ B. B. v$ v$ v$
```

[10] https://icfpcontest2020.github.io/
[11] https://icfpcontest2014.github.io/

Or in human-readable form:

$$(\text{“solve lambdaman6 ”} \mathbin{.} ((\lambda f.\ (f\ (f\ (f\ \text{“RRRRRRRR”})))) (\lambda x.\ ((x \mathbin{.} x) \mathbin{.} x))))$$

We also generated some of the lambdaman puzzles by choosing a small ICFP program that generates a big maze, hoping that the contestants would somehow figure out what the small program was. In fact, the contestants would generally come up with even shorter solutions than ours. One particular example was a maze generated pseudorandomly by a LCG, where we intentionally chose small (two-digit) constants so that if contestants guess that the general formula of the path is LCG-shaped, they could exhaustively search and find the right constants. However, the contestants eventually found different constants that led to a shorter program, used a different fixpoint combinator and a shorter encoding of directions, so their solution was shorter than ours.

4.4 3D

Our favorite part of 2006 ICFPC[12] was the so called 2D language, in which a set of relatively standard programming exercises had to be solved with an ASCII-art based esoteric programming language. For this year we wanted to create a subtask with a similar vibe, so calling the language `3d` was an obvious choice. So we had the name, but what's its definition?

We thought about it a lot, and came up with various ideas:

1. having 3-dimensional operators with 3 inputs and 3 outputs
2. some very non-orthogonal set of basic operators
3. arrange them in smallest 3d space
4. but maybe the length of travel between the operators should matter somehow, too
5. perhaps you could reverse-engineer the internal structure from inputs/outputs

After a deep dive into linear algebra and combinatorics, it turned out that some of these ideas required more than 3 dimensions to work out, and we did not want to go there. Maybe 2 dimensions were the sweet spot after all.

Eventually we started to toy with the idea of “mirroring” an infinite number of copies of a 2-dimensional board above it and below it, where each layer would be shifted by a constant amount along each axis. Then besides running operators within one board, you could invoke the operation “write 1 layer upward”, which would write to the adjacent field of the upper board, and the value would appear in the field offset by (d_x, d_y). Taken further, you could “write n layers upward”, which would manifest as the value appearing at offset (nd_x, nd_y) in the same board.

The change that created the time-travel-oriented language in the way it finally ended up, was stacking the board below its own past states rather than its own clones, and allowing the “write upward” operator to be parameterised by d_x and d_y as well as the number of layers d_t. We discussed a lot if we should keep d_x, d_y fixed or let the programmer choose d_x and d_y; whether we should allow writing into the future, whether to allow $d_t = 0$, etc.

So, in short, the final `3d` language as presented in the contest was as follows. A program is a 2-dimensional grid, where operators move (unbounded integer) values around the grid. For example, operator `>` takes the value from the cell on the left and moves it to the cell on the right in the next timestep. Operators such as `+` only reduce when values are available both on the left and above it, and write the result to the right and below it. For example, a program could go through the timesteps shown in Figure 1.

One special operator is `@`, the time travel operator, which can send a value back in time d_t steps with an offset of d_x and d_y. This operator can therefore be used to create a loop, besides making literal loops in 2D space. This is not only useful for creating correct solutions, but also for achieving

[12] http://www.boundvariable.org/

```
[t=1]       [t=2]       [t=3]
1 > . .     . > 1 .     . > . .
. 2 + .     . 2 + .     . . + 3
. . . .     . . . .     . . 3 .
```

Fig. 1. Example reduction of a 3d program

a low score; the score is the three-dimensional "spacetime" volume of a program, so by using time travelling the time dimension can be artificially kept low.

The full 3D subtask description can be found here[13].

4.4.1 Problems. The contest features 12 different problems to be solved with the `3d` language. Most of these are quite standard tasks that would be quite trivial in regular programming languages, for example the factorial function, the absolute value of a number, the sign of a number, the maximum of two numbers, the least common multiple of two numbers, a primality check, a palindrome check, and matching parentheses. One of the extra challenges with `3d` is that there are no comparison operators, only equality checks. The harder challenges were finding the smallest number B such that the given number A in base B is a palindrome, finding the number of unique squares visited in a `lambdaman` path (which is encoded in a big number), and finally computing the sine of an input number up to a certain precision (`3d12`).

4.4.2 3d12. There was a lot of discussion around `3d12`, the sine problem. Jeroen thought it was hard and suggested making it easier, Matus thought it was easy. In any case, it was a very interesting problem, in several respects.

The task was to compute $\mathrm{truncate}(\sin(A/10^9)\times 10^9)$ for any input number A where $-1\,570\,796\,327 \le A \le 1\,570\,796\,327$. In other words, the task was to compute $\sin(A \text{ radians})$, where the input and the output are expressed in the fixed-point representation with the scaling factor 10^9.

Error tolerance. One interesting change to the problem spec we had to make was allowing an error of 1. (Note that "error of 1" is different from "error of 1 in the least significant digit", as it may affect more significant digits, too.) This might appear as an ad-hoc simplification but it was not. The problem is that the correct truncation is tricky to determine if the number ends with digits $000\ldots$ or $999\ldots$ because then an arbitrarily small error can flip the result of truncation. One might think that simply evaluating the sine to a couple more digits should solve this problem but since `truncate` is not a continuous function – and therefore not computable – there is no theoretical bound on how many extra digits are necessary.

In practice, we allow less than 10^{10} possible inputs (and test even fewer), so there *is* a finite upper bound on the extra precision: we could theoretically iterate over all of them and for each input determine the number of extra digits that's necessary. However, this is a programming contest, not a numeric analysis contest. We therefore decided that instead of having the contestants calculate all these things, we simply allowed an error of 1, which sidestepped all these issues.

Hence this problem relaxation comes from the incomputability of `truncate`. Allowing any larger error (for example to avoid the need to calculate extra digits of precision) *would* be an ad-hoc simplification, though.

Also see: Table-maker's dilemma[14], coined by William Kahan:

> Nobody knows how much it would cost to compute y^x correctly rounded for every two floating-point arguments at which it does not over/underflow. Instead, reputable

[13] https://github.com/icfpcontest2024/icfpc2024/blob/main/static/3d/3d.md
[14] https://en.wikipedia.org/wiki/Table-maker%27s_dilemma#Table-maker's_dilemma

> math libraries compute elementary transcendental functions mostly within slightly more than half an ulp and almost always well within one ulp.

Above, "ulp" is "unit of least precision", i.e. the order of magnitude of the last digit. Kahan talks about y^x but sine is transcendental as well.

Also, Kahan talks about rounding but that's as incomputable as truncation. We decided to use truncation because it does not differ in computability, but it's much easier to communicate to the contestants: there are many strategies of rounding but only one notion of truncation.

Eliminating rounding errors. When working off the Taylor series of sine with terms up to n-th order, one may notice that all denominators divide $n!$. Therefore, they can be placed on a common fraction line with denominator $n!$, where the numerator contains only integers, and can therefore be error-free. The division can be performed only once, as the last step of the computation, and the entire error of the computation is due to the truncation of the Taylor series.

Performance. But we don't even need to evaluate $n!$. Another interesting property of this problem is that it's actually computable quite efficiently: one doesn't need more than 8 iterations of computing with numbers scaled by 10^{10}, which is one digit of precision more than the problem statement requires. Using Horner's method, the series can be evaluated without ever calculating $n!$. Starting from:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

Horner's method tells us to add brackets:

$$sin(x) = x \cdot \left(1 - \frac{x^2}{2 \cdot 3}\left(1 - \frac{x^2}{4 \cdot 5}\left(1 - \frac{x^2}{6 \cdot 7}(1 - \dots)\right)\right)\right)$$

Taking the first 8 terms of the series (up to and including the 17-th order), this yields the following sequence:

$$\begin{aligned} a_{19} &= 1 \\ a_i &= a_{i+2} \times x^2 / i / (i-1) \times (-1) + 1 && \text{for } i < 19 \\ \sin(x) &= a_3 x + \text{error not larger than } \frac{x^{17}}{17!} \end{aligned}$$

When using the multiplier of 10^{10} and 8 terms in the Taylor series, the maximal error of this sequence due to truncation after division in every step comes out to no more than 2.4×10^{-10}. This is sufficient to ensure that the total error after truncation, including the Taylor series error, will not exceed 10^{-9}.

Shape of time. While designing the 3d language, the shape of time went through some significant evolution.

The time-travelling 3d was originally conceived as concurrent processes forked at different points in time of the evolution of a single board. The computation would start as a single process evolving one board, forming a linear history of board states. Each warp operation would fork a new thread of evolution branching out from that past state, running concurrently with the original thread that would continue the original timeline.

Once the program spawns multiple threads like this, these can write into each other's boards, i.e. perform interprocess memory writes, and with a bit of organisational effort, one could write a coroutine-shaped program this way. For example, you could write a program that would receive a sequence of values sent from the future over several time steps.

At first, we defined that the threads mutate a single linear sequence of boards as their state storage, $\mathrm{st}_{t+1} \leftarrow \text{evolve } \mathrm{st}_t$. This has the advantage that time, which you need to give to warp operations, and which identifies the target board of a timewarp write, is a simple integer. You can also write into the future of a board, given that you've previously warped far back enough. The disadvantage was that we needed to specify what happens on conflicting writes/warps, and there were quite a few ways to cause conflicting writes/warps.

All that becomes easier when we switched to a multiverse shape of time, where the board states form a tree: each branch is forked at the destination of a timewarp, and evolves linearly until a timewarp into the branch causes further branching. The rules become a lot simpler because warp conflicts are avoided but you have to deal with the fact that boards are now identified by a path down the multiverse tree rather than a simple integer. This also means that it's impractical to write/warp into other branches or forward, as the board identifier would have to be something more complex than an integer. An integer lets you only write backwards in timeline of the same branch.

A nice aspect of the multiverse model was that pretty much all programs that worked with the linear model of time worked unchanged with this new model as well. We hoped that the contestants would be able to use and abuse the multiverse nature of the time, and to come up with tiny solutions that would be beyond our imagination.

However, explaining the whole thing precisely enough that the contestants could implement their own interpreters required a lot of words, and we decided to simplify the language to the minimum that would make an interesting challenge. Hence we removed the concurrency aspect of the language, and decided there would be only one thread of execution, and if you warp back, your future is halted and erased instead of kept running. That's the `3d` language as stated in the problem description. We were a bit worried that this would "vulgarise" the time travel operator to a simple GOTO but its interaction with time was interesting enough, and the problems it brought were novel and annoying enough (as described by various teams), that we decided to keep it that way.

4.5 Efficiency

The last task that contestants were presented with was the so-called efficiency task. The story is that the Cult of the Bound Variable has a quantum computer to evaluate complex expressions, but to limit energy consumption contestants must learn to evaluate these in another way. Each of the problems for this subtask is an ICFP expression which will (eventually) evaluate to an integer, which contestants must submit. So, the scoring for this subtask was binary.

However, the name of this subtask was purposefully a bit misleading, as it is not really so much about evaluation efficiency, but rather about reverse-engineering. The following sentence from the problem statement hinted at this:

> However, students may need more than just an efficient evaluator in order to get the right answer (and likely don't possess a quantum computer).

The intended solution was to pretty-print the given expressions, and potentially re-implement parts of the expression in a more efficient way, or maybe in a different programming language.

4.5.1 Problems. The problems for this subtask were set up in roughly increasing difficulty. To make things even more confusing, `efficiency1` *could* be evaluated with an efficient evaluator, and with enough patience `efficiency2` and `efficiency3` potentially as well, but from then on evaluating became infeasible. For example, `efficiency4` computes the the 40-th Fibonnaci number in a naive way, which would take in the order of 2^{40} evaluation steps:

```
B$ B$ L" B$ L# B$ v" B$ v# v# L# B$ v" B$ v# v#
L$ L% ? B< v% I# I" B+ B$ v$ B- v% I" B$ v$ B- v% I# II
```

In a human-readable form:

$$
\begin{aligned}
&(((\lambda v_1.\ ((\lambda v_2.\ (v_1\ (v_2\ v_2)))\ (\lambda v_2.\ (v_1\ (v_2\ v_2))))))\\
&\quad (\lambda v_3.\ (\lambda v_4.\ (\textbf{if}\ v_4 < 2\ \textbf{then}\ 1\ \textbf{else}\ ((v_3\ (v_4 - 1)) + (v_3\ (v_4 - 2)))))))\\
&\quad 40)
\end{aligned}
$$

Squinting your eyes, the Fibonacci function can be recognized, but a possibly easier way of solving this is by replacing the constant 40 (which is `II` in the original expression) by 1, 2, 3, 4, etc., in which case evaluation is feasible and then the sequence can quickly be recognized. The same approach worked for several other problems.

4.5.2 Wrong answer. Unfortunately the organizers don't actually posses a quantum computer, so in order to prepare this problem we had to fall back the the same approach as contestants. Of course knowing the construction of the expressions makes it easier to reason about them, but in many cases we could not actually verify our answers.

This led to our answer for `efficiency8` being wrong. This problem encodes a 3-SAT problem with 50 variables and multiple solutions, and the expression iterates over a number whose bits represent the values of the variables, until a satisfying assignment for the 3-SAT problem is found. However, we mixed up endianness of our number, so our answer was the smallest integer if you would reverse the bits. Fortunately, one of the teams found this out and reported this so that we could correct our answer.

4.6 Endo

ICFPC 2007 was another edition we wanted to take inspiration from, which features the alien Endo who stranded on earth and tried to get back to its planet. For a long time during the preparations of this years contest we had a subtask named `endo` on the list, but our idea did not completely work out, and due to other subtasks ending up in a better state, we decided to remove `endo` from the list. To our excitement, some teams did try to `get endo` during the contest, in hindsight we could have hidden a small easter egg there.

5 UNLOCKING AND BACKDOORS

The four real subtasks were not directly visible to contestants at the start of the contest. To bring some element of surprise into the contest, we decided to go for a setup where these were 'unlocked' step by step.

At the start, only the `hello` subtask was visible, and then contestants had to score points for `hello2` (doing `get scoreboard`), `hello3` (using the echo service) or `hello4` (solving the language test) before getting access to `lambdaman` and `spaceship`. Once at least 5 problems of both `lambdaman` and `spaceship` were solved, the `3d` subtask was unlocked, and finally after solving 5 of the `3d` problems, teams got access to `efficiency`. To be inclusive for less experienced teams, we decided to unlock everything for all teams after the first 24 hours of the contest.

On top of this, we decided to hide some easter eggs / backdoors to unlock `3d` and `efficiency`. Specifically, the problem for `lambdaman21` has contains a big number which is constructed as (very big number) + 4025597445063108291862138 8, and if you interpret that number as a string (as with the `$` unary operator), it spells `3d-backdoor`.

Finally, in `spaceship22`, which looks like a big lambda if you plot it, we hid another backdoor. In the beginning of the expression there is a small piece of string being thrown away, which spells `/etc/passwd`. If you do `get /etc/passwd` you see the username and md5 hash of a password of the headmaster, and that could get you into the `efficiency` course.

6 SCORING

Because we went for a setup with different subtasks, scoring was challenging. The different nature of the subtasks made this even harder, for example:

- Some tasks go from 0 points to 1 point (`efficiency`), more is better;
- some tasks go from 1M to potentially tens of points, less is better (`lambdaman`, `spaceship`).

For the most fun contest, we want all subtasks to fairly and equally influence the overall result, so that it is impossible to be super good at just one subtasks and win the contest with that. In the ideal world, the winning team would score well across the board. However, just adding the points won't work because of the wild differences in the nature of the problems, point ranges, numbers of subproblems, and potentially scoring distributions.

At first, we came up with multiplying the `efficiency` scores by 100 to bring them roughly in the range of the other problems, but that wasn't going to work well. A different number of problems per subtask and a potentially different difficulty of each subtask would still skew the overall scores.

Another idea was to make some assumptions about the distribution of ability among the contestants, and created a scoring system based on the p-value of a team's ranking given the null hypothesis that the team is average across all tasks. After some algebraic simplification, this was equivalent to ordering by the product of ranks across subtasks.

That was behaving better but it was too sensitive at the top of the scoreboard: if you improved your rank from 2 to 1 in some subtask, your overall score would as much as *double*. The results were then a bit unpredictable. Also, our assumptions about the score distribution probably weren't completely sound.

We also made a Scoring Lab, which was a small web tool that calculated the scoreboard using all approaches we considered to see how they behave. This way, we could look for interesting examples of scoring behaviour and send them to each other to discuss.

The crucial insight was that we could interpret the scoring process as an election, where the teams are the candidates, and the problems/subtasks are the voters. This invites the use of the Kemeny-Young preference aggregation[15]. The downside of this is that this method is not tractable for hundreds of teams; the only algorithm we were aware of runs in $O(n!)$ time.

We consulted Prof. Georgios Gerasimou, an expert in decision theory, and he suggested looking at the Borda Count[16]. After putting it in Scoring Lab, we decided that the behaviour of Borda Count looks the best of all the methods we've seen so far. In the end this is what we went for, and in the evaluation we held after the contest this was mentioned several times as a positive element of the contest. Borda also nicely deals with ties: eventually in `hello` and `efficiency`, everyone from the Top N was tied in the first place, which meant that these subtasks naturally stopped influencing the scoreboard.

During the implementation of the Borda count, we had some discussion on whether we should run a single Borda aggregation over all problems, or whether we should first aggregate problems per subtask and then subtasks into the contest board. We went with the latter, as it does not give more advantage to subtasks with more problems.

6.1 Scoreboard

There was a lot of discussion how much scoring we want to disclose on the scoreboard:

- Do we want to disclose only the global (contest-level) ranks?
- Do we want to disclose only all the ranks and have per-subtask scoreboards as well?

[15] https://en.wikipedia.org/wiki/Kemeny%E2%80%93Young_method
[16] https://en.wikipedia.org/wiki/Borda_count

- Or even per-problem scoreboards?
- Do we want to disclose all the scores?
- Do we want to disclose the best score for each problem? ("You submitted a solution scoring X, the best score is Y.")

The organizers did not all agree, for example some suggested that a bit of mystery makes the contest more fun, e.g. if you can't know if you're underperforming on a particular problem, while others found it important to be as transparent as possible. In the end, we settled for a per-problem scoreboard displaying only the ranks, and also disclosing the best score for each individual problem, so that there is some mystery, but team can still know where they can improve and by how much.

7 SERVER INFRASTRUCTURE

Next to designing an interesting problem, we had to set up the server infrastructure to run the contest. We did have some statistics from earlier years on the number of teams and submissions, but given that this years problem has a different nature (each message sent is a 'submission'), we found it hard to estimate the necessary server capacity. In previous years, AWS Lambda was used, which has the benefit that it automatically scales, but the downside is that the constant overhead is quite large (this page[17] reports 60–90 ms per execution for Haskell), which does not fit our problem since many 'submissions' just get static data and could be handled within a few milliseconds of real runtime. Furthermore, we had a limited budget and costs can grow quickly with AWS.

Since the two main organizers have many years of experience running the backend of chordify.net[18], we decided to use what we know, and use the same software stack. This stack uses Haskell for the core infrastructure, with Warp[19] for the webserver part, persistent[20] for interacting with a MariaDB database, amazonka[21] for interacting with the object storage, and redis-schema[22] for Redis interaction including a distributed worker setup.

The data storage was split up between MariaDB and an Object Storage. The first is convenient for storing almost all of the contest data (teams, submission metadata, scores), but decided to keep the complete messages as well, which is data that can grow quickly in size. So, we stored the ICFP messages (both request and response) on the object storage with their md5 hash as identifier, and stored only this hash in the database.

The main Haskell webserver handling all requests stored the request information, and then using the redis-schema library put a job in the worker queue. Separate worker machines handled these jobs, wrote the response back, which was then returned to the contestants. In this way, we managed to run the whole contest from a set of DigitalOcean VPSs[23], with 5 (single-CPU) worker machines in the beginning of the contest, and 7 later on. To be able to do potential maintenance and upscaling, we had set up a database replica as well, which we used for taking frequent database backups. It turned out the 4-cpu primary replica was powerful enough to run smoothly for the whole contest.

7.1 Limits

To make sure that our servers would not get overloaded, we needed to put limits on the evaluation of the ICFP expressions that contestants sent. An easy choice would be to limit the wall-clock time,

[17] https://theam.github.io/aws-lambda-haskell-runtime/
[18] https://chordify.net
[19] https://hackage.haskell.org/package/warp
[20] https://hackage.haskell.org/package/persistent
[21] https://hackage.haskell.org/package/amazonka
[22] https://github.com/chordify/redis-schema?tab=readme-ov-file#remote-jobs
[23] https://www.digitalocean.com/

but we felt that this was not a good metric, since it is influenced by the efficiency of our evaluator (which we did not want to publish), and the specific hardware used. Instead, we decided to pick a deterministic metric, which is the number of beta reduction steps, and only use wall-clock time and memory limits as a fallback.

Unfortunately, we underestimated the contestants, and some managed to submit ICFP expressions that would stay within the beta reduction limit but still run into the time limit. This seemed to be mostly due to big integer operations; the ICFP language does not limit the size of the integers, but unsurprisingly, arithmetic operations do get slower when you perform them on large numbers. To deal with this, we decided to change the server setup mid-contest: whenever the wall-clock time limit or the memory limit was reached, we reran the submission on a more powerful machine with larger limits. The benefit of this setup, in in contrast with increasing the limits for all workers, is that a bunch of slow expressions would not overload the regular workers as these would cut them off quite quickly, so we always had workers available for the easy expressions (e.g. getting a problem). The 'slow queue' was full from time to time, but with just two workers and a 10-second timeout, there was little waiting time for contestants, and the hard limit was reached less frequently than earlier in the contest.

7.1.1 Limits in 3d. For the 3d problem, we needed some extra limits, since we had multiple layers of evaluation: an ICFP expression to be evaluated, which resulted in a 3d program to be evaluated on a set of inputs.

Tick limit. We imposed the tick limit in the same way as the beta reduction limit for the ICFP language: the number of next-board-state operations we are willing to compute while evaluating a submission. That's a straightforward limit: we would do at most 10 million board reductions.

Information density limit. The size of the bounding box of an executing 3d program was counted in terms of cell coordinates. But a cell could potentially contain an arbitrary-sized integer and there was the danger that teams would encode a lookup table of precomputed answers in a huge integer, and then build a tiny lookup machine around it. That was against the spirit of the challenge, and we wanted to prevent it.

We therefore limited the information density of the source code to about 8 bits per cell: 199 integers + a couple of operators. This limitation applies only to the "source code" of the boards, i.e. their initial state; afterwards, the programs are free to compute with integers of arbitrary size.

Wall-clock-time limit. The programs can become quite slow: it's easy to square a number in every tick, making it grow exponentially in memory. We could place the tick limit to the logarithm of the maximum allowed integer size to prepare for the worst case, but that would unnecessarily cripple well-behaved programs. Instead, we decided to just count on the already existing wall-clock time limit for stopping these programs. From a manual check, most of the 3D programs running into the time limit were programs that entered an infinite loop.

Test evaluator limit. We provided a service that would reduce any given board a couple of times and sent the trace (board states) back. We wanted to make the contestants write their own interpreters, so we wanted to make sure that the contestants can't just run anything in organizers' interpreter. Therefore this service needed a limit on how many reductions it would compute.

The original limit was tens of reductions. But we realised that in our own programs, warps would mostly use $d_t = 5$ or $d_t = 7$ or such small numbers. That means that the teams could (ab-)use our evaluator to run one warp iteration, and then resubmit the results repeatedly to reduce them to completion. We therefore decided to restrict the number of reductions allowed to 4, which we thought would prevent most useful warp loops, while still being useful for testing the rules

of reduction. (Little did we know that some teams came up with a way of warping after every computation, so their programs never went beyond t=2.)

8 RESULTS

The organizers would like to make some statements about programming languages:

8.1 Winner of the lightning division

Spreadsheet and Rust are very suitable for rapid prototyping.

Congratulations to teams MANARIMO (Spreadsheet) and UNAGI (Rust) for being tied at the first position (for only about 15 minutes) after exactly 24 hours.

8.2 First prize

Rust is the programming tool of choice for discriminating hackers.

Congratulations to team UNAGI (Rust) for winning the contest!

8.3 Second prize

OCaml is a fine programming tool for many applications.

Congratulations to team B$ L$ B$ v$ v$ L$ B$ v$ v$ (OCaml, C++) for taking the second place.

8.4 Third prize

C++ is also not too shabby.

Congratulations to team PURELY FUNCTIONAL NETWORKS (C++, Rust, Python, Haxe) for ending up at the third place.

8.5 Judges' prize

PANICKED ALBATROSS are an extremely cool bunch of hackers.

Team PANICKED ALBATROSS (Python, C) was picked by the jury because they were just a 2-person team, ending up at rank 33, but they did get the absolute best score for problem `3d12`, the most discussed problem in the whole contest. Furthermore, they published a nice writeup[24] explaining how they constructed their `3d12` solution.

8.6 Honorable mention

We would also like to give a honorable mention to team PIGGYHACK for extracting the testcases for some 3D problems, even though we tried really hard to make that infeasible.

9 STATISTICS

Here are some statistics about the competition:

- 471 teams registered (though some of them are duplicates)
- 365 teams sent in at least one message (submission)
- 355 teams got a non-zero score
- 195 808 messages were sent, of which 67 947 scored points
- in the after-contest survey, the question "How much did you enjoy the contest?" (with a 1-10 scale) scored an 8.7 on average with a standard deviation of 1.74

Figure 2 shows a plot of the number of submissions over time.

[24] https://github.com/estansifer/icfpc2024

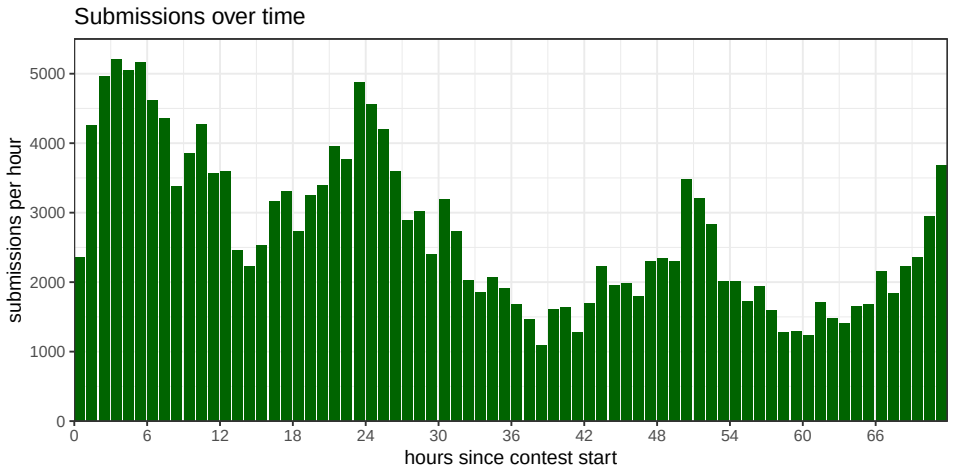

Fig. 2. Submissions over time

Figure 4 shows a plot of the top 33 ranks over time. Each team corresponds to a line of a certain colour. The final positions of the top 33 teams are indicated by labelled dots. Many teams are below the bottom edge of the plot, and thus not displayed, but the plot does contain all awarded teams.

For each team, we've generated such a plot, highlighting the journey of the team. Figure 3 shows an example. These plots were published before the results could be disclosed, so we redacted two hours before and after every key time point (end of the lightning round, end of the contest). All plots can be found on the website[25].

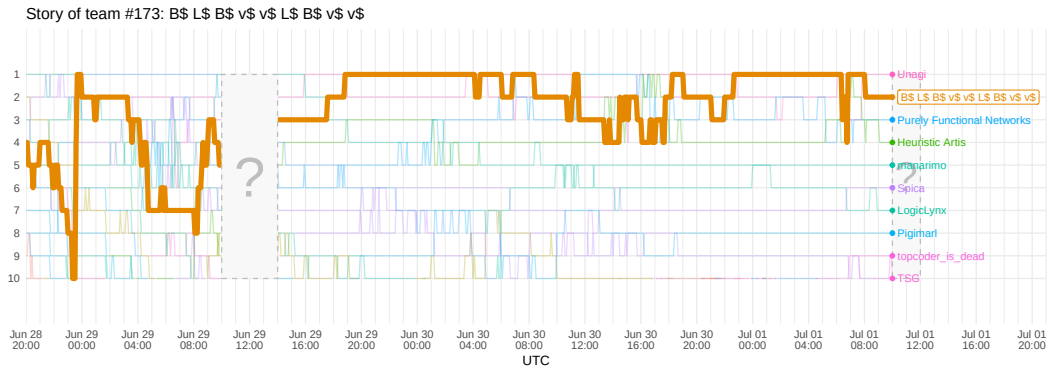

Fig. 3. Personalised story of team #173 B$ L$ B$ v$ v$ L$ B$ v$ v$

[25] https://icfpcontest2024.github.io/journey.html

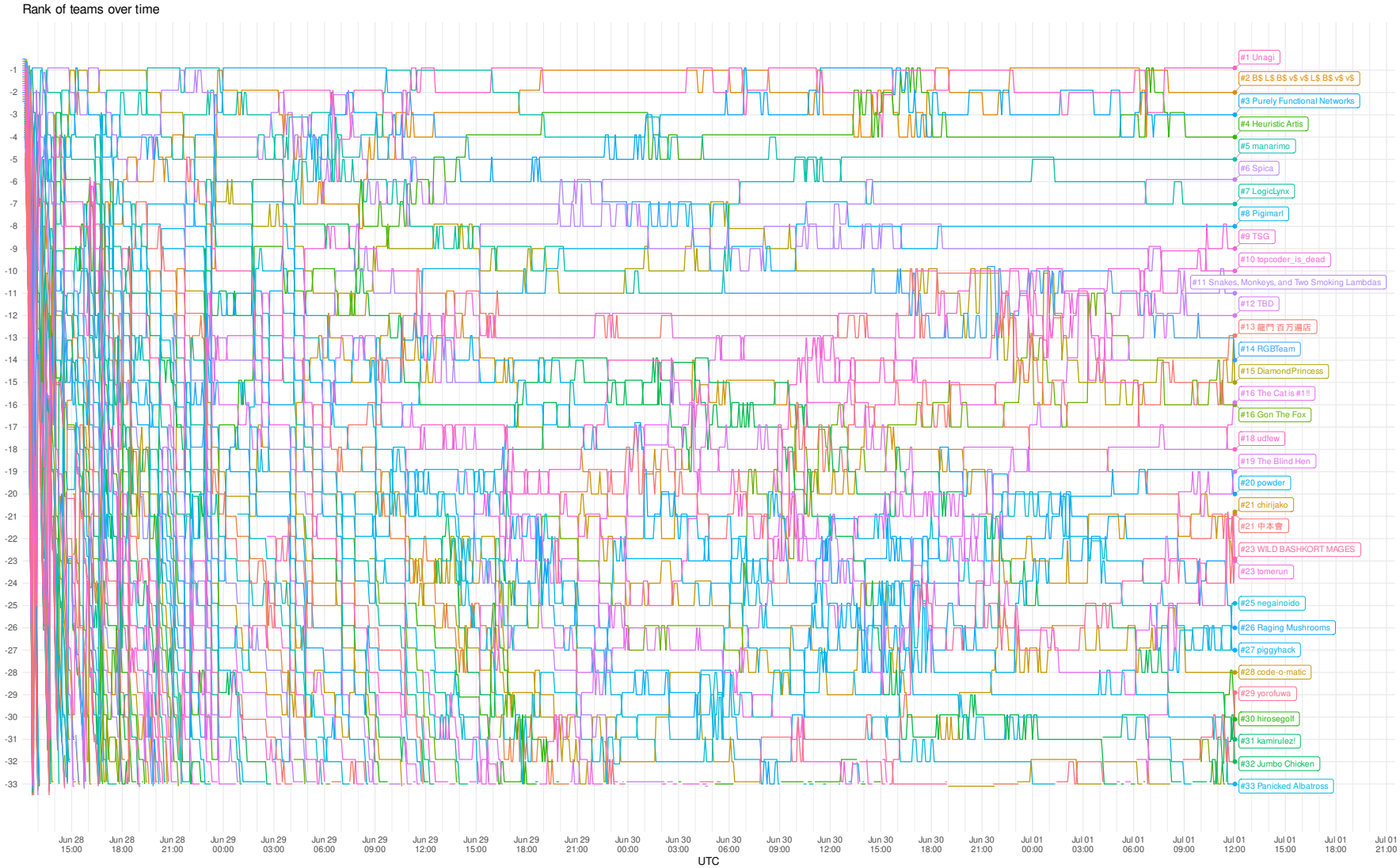

Fig. 4. Ranking journeys of top 33 teams

10 ACKNOWLEDGEMENTS

We would like to thank those that helped making ICFPC 2024 possible:

- Bas den Heijer for inspiration and early brainstorming;
- Aymeric Fromherz for mentoring;
- Apoorva Anand for test solving;
- Prof. Georgios Gerasimou for suggesting the Borda Count for the scoring system;
- ICFPC 2006[26] which has been a huge source of inspiration;
- Internet Problem Solving Contest[27] which has been a secondary source of inspiration;
- CodeWeekend #1[28] from which we stole two test cases;
- The ACM SIGPLAN International Conference on Functional Programming[29] for sponsoring the contest.

[26] http://www.boundvariable.org/
[27] https://ipsc.ksp.sk/
[28] https://codeweekend.dev/
[29] https://icfpconference.org/