

2023 CMIMC Programming Contest Problem Packet

Table of contents:

Page 2. Submission Guidelines
Page 3. AI Problem 1
Page 4. AI Problem 2
Page 5. Optimization Problem
Page 7. New Language Problem





















Submission Guidelines

The CMIMC Programming contest will begin at around 7:30 PM (UTC−4) on Friday, April 23, 2023. The contest will end at around 4:00 PM (UTC−4) on Sunday, April 25, 2023.

General problem instructions:

- Submissions to the CMIMC servers will open 12 hours after the start of the contest.
- For each problem, each team may submit once every 15 minutes to the CMIMC servers. Only the most recent submissions are kept.
- Each submission must be less than 500 kB in file size.
- Submissions can only use one process. If more than one is used, every subsequent output will be considered invalid.
- CMIMC organizers may decide to change the number of games played indicated below

Lebomb James (AI Round)

- Every 5 minutes, games are ran. If a team just updated their submission in the past 5 minutes, it is guaranteed that their new submissions will get played at least 10 times. Submissions not just updated will also be used in random pairing.
- Teams will receive the entire game data for all games they played in.
- After the contest has ended, another 100 games will be played for each player. Final results will be computed from these games.

Auction House (AI Round)

- Tournaments are run every hour, with all players participating in one big tournament. A total of 2000 tournaments will be played.
- Teams will receive data about all information their bot would receive, including their own bets, bets of all players they faced, and results.
- After the contest has ended, another 5000 tournaments will be played. Final results will be computed from these tournaments.

Two Maze (Optimization Round)

- Submissions will be run on 30 test cases, and raw scores (times) will be available. A live leaderboard will be present.
- For twomaze, after the contest has ended, submissions will be run on 150 test cases, and final results will be computed with these test cases.

L3 (New Language Round)

- Submissions for each task will be run on 10 test cases, and raw scores (area) will be available. A live leaderboard will be present.
- For twomaze, after the contest has ended, the final results will be computed.

Strange Auction House

God help us if we ever take the theater
out of the auction business or anything else.
It would be an awfully boring world.
- A. Alfred Taubman

Wandering through an enchanted forest, you stumble upon a mysterious house. Unbeknownst to the curiosities within it, you enter and find yourself in the strangest of auction houses in which the bidders are set upon determining who is the strongest of them all through a tournament of auctions. In this tournament, all bidders begin with 100 mana (the forest's currency) in their wallet. In every round of the tournament, each surviving bidder will be randomly pitted against another bidder in a single auction for a fruit of the forest. Both bidders reveal their bids for the item at the same time, and both will have their bid amount deducted from their wallet. The bidder with the lower bid value, deemed weak and unfit by the others, is marked with a loss on their record. Once a bidder reaches five losses, they are shamefully disqualified from the tournament. If two bidders engage in a tie, a coin flip is used to decide the winner of the round—mana is still deducted from each wallet. The last remaining bidder is considered the greatest of them all.

Determined to make a name for yourself in the auction house, you enter the tournament. Every bidder will create their own `strategy` function which determines how much they bid within every auction. Multiple tournaments will be run—mana is reset between each tournament. In the event that a given round of a tournament has an odd number of participants, bidders not matched to an opponent will be given a free ticket to the next round.

You are to implement the function `strategy` that takes in parameters:

- `wallet`, a positive integer value representing the current amount of mana in your wallet at the time of bidding
- `history`, a list of tuples representing your opponent's bidding history in the current tournament. Earlier elements correspond to bids made earlier in the tournament. The first index in the tuple is an integer value representing the bid value made by the opponent. The second index in the tuple is a boolean value representing whether the opponent won the auction (`True` for win, `False` for loss). If a bidder receives a free ticket to the next round, the round will not be accounted for in their history.

`strategy` is to output:

- A non-negative integer less than or equal to wallet representing the amount of mana you wish to bid in the current auction—illegal bid values (negative or exceeding wallet) will immediately disqualify you from the current tournament

Your score will be calculated as follows. If x represents the number of competitors remaining in the round immediately after you are eliminated, then $1/(\sqrt{x} + 1)$ will be added to your score. If no competitors are remaining, or you are the sole bidder remaining (i.e. you are never eliminated), 1 will be added to your score. The higher your score, the better.

Lebomb James

If the present were not so horrible and grim,
and the future so mysterious and enigmatic,
one could go mad with joy.
- Konstantin D. Kavelin

As a warlord in the post-West Russian War, survival is of the utmost importance. Every few weeks, Luftwaffe bombers from the airfields of Reichskommissariat Moskowien fly over the warlord states of western Russia, gleefully bombing any civilian targets they find as a demented form of "pilot training". While they make life hell for rural communities and can occasionally slip through and rain fire upon provincial cities, their lack of reconnaissance means that they will only strike the largest cluster of targets.

Your task as an officer is to establish settlements in the ruins of western Russia. On a 10x10 grid, 5 officers (including yourself) can each place down 3 settlements every turn. Settlements can overlap and you can place multiple settlements in each location. At the end of a turn, the Bomber (controlled by Reichskommissariat Moskowien) will greedily release, one by one, 2 Sternenbomben centered at any grid that will maximally destroy the settlements. If multiple target locations result in the same amount of destruction, the Bomber will choose randomly between the locations. The Sternenbomben destroy all settlements in a cross shape (consisting of five squares). The number of standing settlements from each officer is recorded.

There is a 64MB memory, 30s CPU time, and 3s per query limit. If your program exceeds any of the limits, the settlements will be placed at random locations.

You are to implement the function `strategy` that takes in parameters:

- `pid`, a non-negative integer value corresponding to your player ID, randomly assigned between 0 and 4, inclusive, and fixed throughout the current game.
- `Board`: a 3D array where `Board[x][y][pid]` represents the current number of standing settlements placed by officer `pid` at grid position (`x`, `y`).

Your function does not need to be pure/stateless- you can write code outside of your function and global variables will be saved across rounds.

`strategy` is to return

- A length-3 list of 2-tuples of integers, where each element in the list corresponds to the position of one of the settlements you intend to place down. Each integer should be between 0 and 9, inclusive. If your output is invalid, your settlements will be randomly placed.

The game will be run for 1000 turns, where each officer places 3 settlements simultaneously on each turn. The last 100 turns will be used to evaluate the performance. Your score will be the average number of standing settlements in the last 100 turns, normalized by having the sum of five officers' scores to 5. The higher your score, the better.

Two Maze

"To be alive is to be dizzy and not to know exactly where to go."
— Ander Monson

In this optimization game, you have a robot navigating through an 32x32 integer lattice maze. However, the robot's sensory/movement control system lost synchronization. The Carnegie system can see nearby horizontal maze walls, but can only move the robot vertically. The Mellon system can see nearby vertical maze walls, but can only move the robot horizontally. There is a single method of memory storage and communication between the two systems, which is incrementing an internal clock by an integer number of milliseconds, at least 5ms.

A maze is generated as edges on a 32x32 integer lattice. The robot starts within the square bounded by (0,0)-(1,1), with the clock starting at 0. Your goal is to move the robot to the square bounded by (31,31)-(32,32) while minimizing the final value of the internal clock.

The functions `Carnegie` and `Mellon` will be called alternately, starting with `Carnegie`, until the robot makes it to the destination or when the robot hits a wall. If the robot hits a wall, or 4096 calls to Carnegie and Mellon have elapsed without the robot reaching the end, the robot will have failed. All edges on the boundary of the 32x32 square are walls (so the robot may not leave the 32x32 square), and there are no walls outside.

You are to implement two functions, `Carnegie` and `Mellon`. `Carnegie` takes in parameters:

- The position of the robot (x, y) where $0 \le x,y < 32$, which represents the robot being inside the square bounded between (x, y) and (x+1, y+1).
- Nearby horizontal maze walls, a 16x16 zero-indexed array, entry [i][j] containing 0 corresponding to no wall and 1 corresponding to a wall on the edge (x+i-8, y+j-8) - (x+i-7, y+j-8).
- An array of positive integers containing all past increments of the internal clock.

`Carnegie` is to output, in a pair:

- An integer corresponding to desired movement in the y direction
- A positive integer, at least 5, to increment the clock.

`Mellon` does the same thing, except that it receives nearby vertical maze walls with entry [i][j] corresponding to a wall on the edge (x+i-8,y+j-8)-(x+i-8,y+j-7) and returns desired movement in the x direction and the increment.

`Carnegie` and `Mellon` must be pure functions, that is, functions that yield no side effects. In particular, accessing variables outside the scopes of the functions are not allowed. The CMIMC organizers will check your code manually.

[**Two Maze** problem statement continued on next page]

There are three different maze patterns, and you can implement different `Carnegie/Mellon` functions for each pattern.

- Pattern 1: Random walls generated with a probability of 20%. A path is guaranteed.
- Pattern 2: Random walls generated with a probability of 30%. A path is guaranteed.
- Pattern 3: Cycleless maze generated by a randomized DFS search.

Scoring will only be based on relative performance. For each maze, the final clock times of contestants will be ranked. In case of a tie, the mean ranking of all tied-contestants is used. Programs that failed will be assigned rank equal to the total number of contestants. Your final score will be linearly interpolated from your ranking, where a ranking of 1 receives a score of 1, and a ranking of the total number of contestants receives a score of 0. The higher your score, the better.

L3

"CLEVELANDD!!"
— Lebron James

L3 is a program that manipulates natural numbers. An L3 program consists of a grid where each square contains a natural number between 1 and 30, and a direction (up, down, left, right). The input to the program is a natural number M, which enters the program on the top-left-most square, traveling downwards. M moves around the grid and is manipulated with these rules:

On a single step of the program, suppose that M is on a square with number A and direction X.

- If M is moving in direction X, M is multiplied by the A, and moves to the square in direction X.
- If M is moving in any of the three other directions, if M is divisible by A, M is divided by A, and moves to the square in direction X. If M is not divisible by A, M is unchanged, and moves to the square in the direction opposite of X.

The program outputs when M moves out of the bottom-right-most square downwards.

Actual L3 code is formatted in a comma delimited csv file. It is suggested that you use a spreadsheet tool to write your code, such as Google Sheets or LibreOffice Calc (Microsoft Excel may export some extra invisible metadata, so you may need to copy the text from the csv export into a new csv file). Make sure to export exactly the size of the grid you intended, so the location of the output matches with what the interpreter expects. Hint: To help coding, use color highlighting.

Within each cell, you should write the number followed by the direction in any of the following forms: u/d/l/r, U/D/L/R, n/s/w/e, N/S/W/E. Additionally, you may add a ";" at the end to indicate a watch point.

For readability, you may leave squares empty, and an error will occur if M moves onto an empty square. Similarly, if M moves out the grid not in the bottom-right-most square downwards, an error will occur. Assume reasonable constraints on the maximum number of steps a program may execute (currently 20000) and the maximum size of the grid (currently 100x100).

1R	2L	1D

Example A - clear register. Input = 2^x. Output = 1.

1D	1L
1D	3U
1R	2U

Example B - transfer register. Input = 2^x. Output = 3^x.

[**L3** problem statement continued on next page]

Your task is to implement the following programs in L3.

1. `Addition`: Input = 2^x3^y . Output = 2^{x+y}.
2. `Comparator`: Input = 2^x3^y . Output = 2 if $x > y$, 3 if $x < y$, 1 if $x = y$.
3. `Multiplication`: Input = 2^x3^y . Output = $2^{x \cdot y}$.
4. `Integer division`: Input = 2^x3^y (assume $y > 0$). Output = $2^{x\,//\,y}3^{x\,\%\,y}$.
5. `GCD`: Input = 2^x3^y (assume $x, y > 0$). Output = $2^{gcd(x,y)}$.
6. `Square root`: Input = 2^x. Output = $2^{floor(\sqrt{x})}$.

It is not hard to argue that L3 is Turing-Complete. However, it is very tedious to deal with streams of data in L3, and encoding streams of data is terribly inefficient. The language L3 extended (L3X) aims to solve this issue.

In L3X, multiple numbers may be manipulated by the program simultaneously. Besides containing natural numbers, a grid may contain a '%', '&', or '~', still with some direction X.

- On a '%' square, M is duplicated, where one M travels in the direction of X, and one M travels in the opposite direction.
- On a '&' square (with direction X), when a number M enters the block in direction X, it is stored in a FIFO queue unique to the square. When another number N enters the block in direction not X, the number N is multiplied with the first element of the queue, M, M is removed from the queue, and the product NM travels in direction X. If the queue is empty, an error is raised.
- Finally, on a '~' square, the number M is set to 1 and continues in the direction of X.

There will be a limit of at most 10 different numbers running actively in an L3X (not stored in the queue of any '&' square). Two numbers may never appear on the same square (excluding those stored in the queue of any '&' square), and an error will be raised if that happens.

In any L3 program the grid is indexed by (row, column). If the grid has height h and width w, input to the program enters the grid on square (0, 0) moving in direction (1, 0), and the output of the program exits the grid on square (h-1, w-1) moving in direction (1, 0).

In a L3X program the square at (0, 1) must be a "&" square. Inputs to the program will be a single number N in the top-left-most square, and a sequence of numbers already stored in the "&" block. The program should output a sequence of numbers traveling downwards from (h-1, w-2), and finally a single number downwards from (h-1, w-1).

[**L3** continued on next page]

~E	&S	1E	1S
1S	%W	~N	2S
1E	1E	1S	1S

Example C - move single number from input stream to output stream (special case of task 9).
Input = 2^1 and [2^x]. Output = 2^1 and [2^x].

Implement the following programs in L3X.

7. `Countdown`: Input = 2^n and [ ] (assume $n > 0$). Output = 1 and [2^n, 2^{n-1}, …, 2^1].
8. `Transfer`: Input = 2^n and [2^{x_1}, 2^{x_2}, …, 2^{x_n}]. Output = 2^n and [2^{x_1}, 2^{x_2}, …, 2^{x_n}]
9. `Sum`: Input = 2^n and [2^{x_1}, 2^{x_2}, …, 2^{x_n}] (assume $n > 0$). Output = $2^{x_1+x_2 \ldots + x_n}$ and [ ].
10. `Max`: Input = 2^n and [2^{x_1}, 2^{x_2}, …, 2^{x_n}] (assume $n > 0$).
    Output = $2^{max(x_1, x_2, \ldots, x_n)}$ and [ ] .
11. `Median`: Input = 2^n and [2^{x_1}, 2^{x_2}, …, 2^{x_n}] (assume n is odd).
    Output = $2^{median(x_1, x_2, \ldots, x_n)}$ and [ ].

To test your program, run python l3.py -c code.csv -t testcase. Use the -d flag to enable additional debugging information, and -x flag to use L3X. Test cases are encoded in .json files. Numbers are encoded as a length 10 list corresponding to powers of the first 10 primes. Otherwise, the test case format is self-explanatory.

We will provide you with a few test cases for local testing, and you are welcome to write more test cases. When you submit your code, your code will be tested against a more thorough set of test cases. You should submit code to each of the 11 tasks above separately.

---

w	x
All	2
90%	1.5
75%	1.3
50%	1.2

All 11 tasks are equally weighted in scoring. Scoring is code-golf like. You will receive extra points for writing smaller programs, determined by the area of the rectangular grid of your program. According to the grid on the left, if your area is strictly smaller than at least w% of all valid submissions (or strictly smaller than all other submissions), your score to the problem will be multiplied by a factor of x. Only the best factor is multiplied.

[**L3** end]