CS F372: Operating Systems
1st Semester 2025-26
Assignment 2
Marks: 60

Release Date: 30th October, 2025

Deadline: 13th November, 2025, 23:59:00 hours (no extensions will be granted, use your free late days if you have any left)

Max free late days per student for the entire semester: 3 days. (the 'last updated' timestamp will be treated as the submission date, even if that involves changing only a single character in your submission)

Use Piazza to clear your doubts.

Operational Guidelines

- The assignment is to be done individually
- The assignment needs to be done using the C language. Your solution should be POSIX compliant and run on an Ubuntu System (>= 22.04). You are allowed to use either multiple processes or threads or both for *efficiency.*
- Only the `-pthread` flag will be passed to gcc. There is no guarantee that other non-standard headers or macros will work.
- After the assignment deadline is over, the solutions will be evaluated against a few more hidden test cases to arrive at the final score. The submission will be timed after the deadline and part of the score will depend on how efficient your solution is.
- **Time Limit:** During evaluation, test cases will time out if their execution time exceeds 120 seconds
- **Efficiency Metric:** There are two efficiency metrics in the assignment: the total number of turns taken to solve a testcase, and the total number of expired packages (count of each time a package is delivered after its TTL). Marks for efficiency will be equally split between these two metrics. Your goal is to minimize both while arriving at the correct solution.

- ***All your submissions*** **will be checked for plagiarism (including AI usage). Any hints of plagiarism will attract a summary 0 score. There is no partial dishonesty**
    - Lifting code directly from the Internet, including from Stack Overflow and/or github and/or other platforms including ChatGPT, is plagiarism.
    - Discussions are encouraged, but copying is not. One thumb rule is: after a discussion do not take away any written or soft copy notes. Instead, watch something mind-numbing, and then implement what was discussed from memory. Look at the Cheating Vs. Collaborating Guidelines here: https://www.cse.iitd.ac.in/~mausam/courses/col772/spring2024/
    - The purpose of the course and the assignment is for you to learn, shortcuts might fetch you grades but you will learn nothing and will have to bear the burden of being a cheat.
    - It is your responsibility to secure your code and prevent others from 'stealing' your code. If your code is in the possession of someone else because of your negligence, it is your responsibility.
    - *Whatever risks you undertake, you are adults so you understand the consequences of being caught. Please keep your chin up and accept the consequences.*
- Every student gets 3 free late days in the semester. Beyond the free late days, every additional late day will incur a 10% penalty. Partial days are rounded up.

Problem Statement

This assignment is on efficient scheduling.

You will be given a **NxN** grid and **D** trucks, and in every turn you may move the truck in any of the 4 possible directions (up,down,left,right) or have it stay in the same cell. On set turns, packages will appear on some specific cells, scheduled to be delivered to another cell before it expires. Your task is to write a *POSIX* compliant C program to control the drivers and deliver all the packages in as few turns as possible.

Catch: On each turn, in order to move a non-empty truck you must find a random authorization string whose length is equal to the number of packages in the truck. No string is required in case the truck is empty, or if you don't wish to move the truck in this turn. Empty trucks may also be moved, not requiring any authorization string to do so. The authorization string corresponding to a particular truck can be obtained by communicating with one of the solver processes using a message queue. There will be S such solver processes available (S <= D), with a separate message queue for each.

Each of the packages will also have a *'Turns To Live'* parameter, beyond which the package will be considered expired. The turns taken to deliver a package is the difference between the turn when the package is delivered and the turn at which the **request** for the package has arrived. If a package is delivered post its TTL, it will be considered as expired. You need to **minimize** the number of expired packages you deliver.
All packages need to be delivered to their requested destination, whether they are expired or not.

Each cell can potentially be a *'toll booth'* cell, which requires the truck to wait for a fixed amount of turns before it is able to move again as a fee. If a truck arrives at a toll booth cell, it cannot move for the next X turns, as described by the toll booth. Any instruction to move the trucks in these turns will be ignored by the helper. The toll booth cells will not be given to the student program explicitly. At each turn, how many trucks are waiting in each toll will be communicated. More details are later in the document.

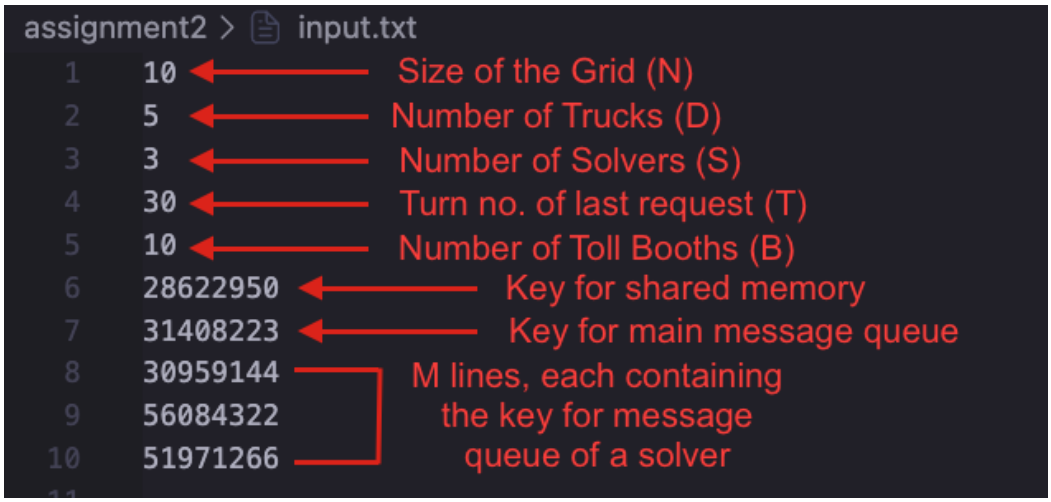
After obtaining all the authorization strings for a turn, you can set them in a shared memory segment.
In the same shared memory, you may also set commands for how you wish each truck to move (move up 1 cell, move down 1 cell, move right 1 cell, move left 1 cell, or no movement), as well as which packages you would like to get picked up/dropped by which truck (these packages will be moved to/from the cell the truck is currently on, before the truck moves this turn). Then, using a main message queue, you may communicate with the main helper process so it can verify your authorization strings and thus perform the changes as requested.

Once the strings have been verified, the helper will reply with the state of the new turn, allowing the process to be repeated.
The helper and solver processes will already be running and ready to communicate with your program. You don't have to write these.

More details for the same are given below.

The Input

Your program will not receive any command line arguments. The initial input will be provided in a file named **input.txt**, which will be present in the same directory as the one your code is executed from. The file will contain the following values:



Constraints:

N <= 500

D <= 250

S <= D

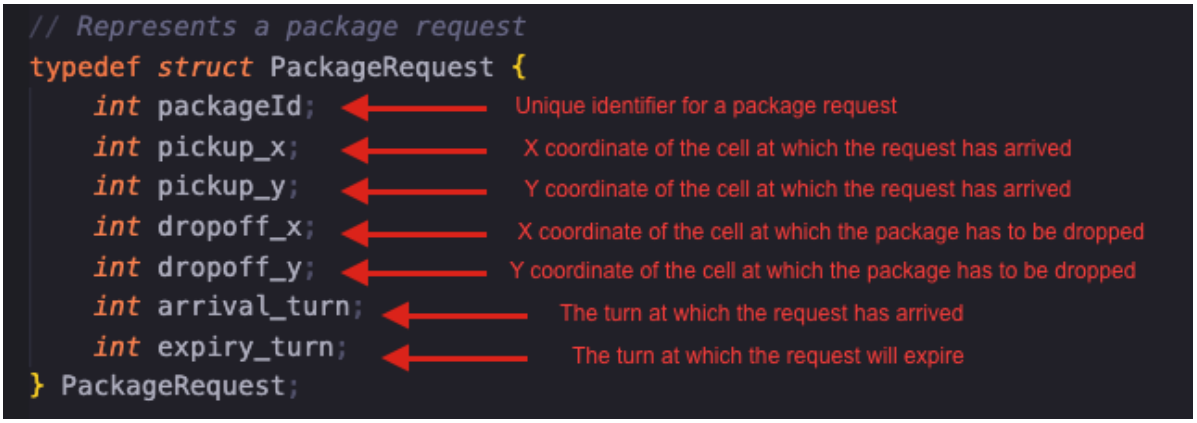
T <= 500

B <= 500

In each turn, no more than 50 new packages may arrive.
Each authorization string will consist only of the letters 'u','d','l','r' of the alphabet, all lowercase.
A truck can hold at most 20 packages.

Package Requests

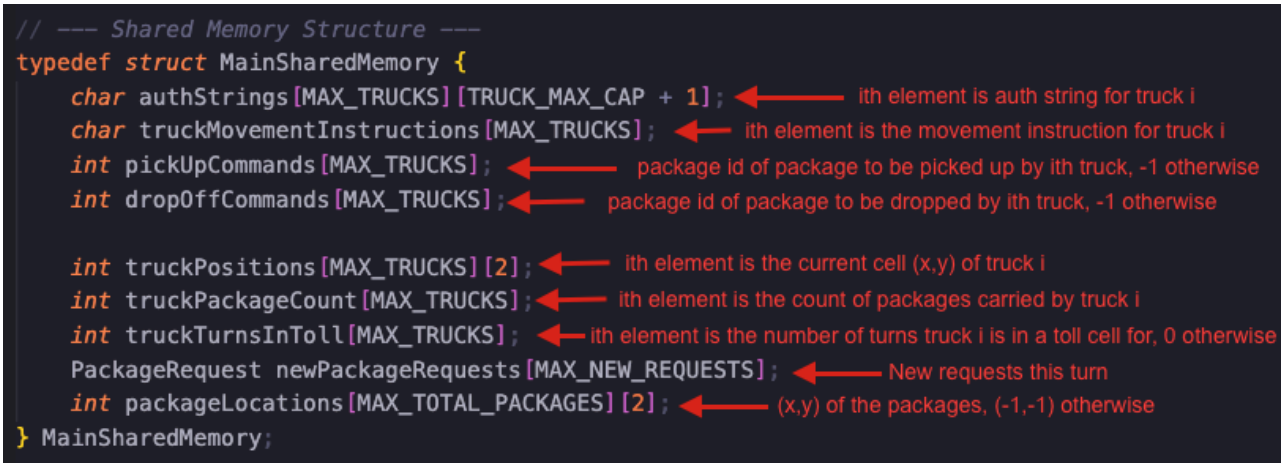
Each package arrival request will be given through the shared memory in the form of the following struct:



The `expiry_turn` holds the value of the run number at which the package request will be expired. If the package has been delivered to its desired drop off cell after any turn greater than this value, it will be considered as an expired package.

The Shared Memory

The shared memory can be modified by both the helper program as well as your program. It will be represented using the following struct:



Here, **MAX_TRUCKS** can be replaced by 250, **TRUCK_MAX_CAP** by 20, **MAX_NEW_REQUESTS** by 50, and **MAX_TOTAL_PACKAGES** by 5000.

Here is additional information about the different fields:

- `authStrings`, `truckMovementInstructions`, `pickUpCommands` and `dropOffCommands` have to be set by your program before each turn. The helper will process these instructions before the next turn. They do not need to be set before the first turn.
- `newPackageRequests`, `truckPositions` and `truckPackageCount` will be set by the helper in each turn. The student program must not modify these.
- In `authStrings`, `pickUpCommands` and `dropOffCommands`, which are to be set by student program, only the first D (0 to D-1 indices), where D is the number of trucks for a testcase, should be used.
- `truckMovementInstructions` only accepts 5 valid instructions, i.e. if the current cell is (x,y) -
    - If the instruction is set as **'l'** (for left), the truck will move to the left cell from the current one, i.e. (x-1,y) in the next turn.
    - If the instruction is set as **'r'** (for right), the truck will move to the right cell from the current one, i.e. (x+1,y) in the next turn.
    - If the instruction is set as **'d'** (for down), the truck will move to the downward cell from the current one, i.e. (x,y+1) in the next turn.
    - If the instruction is set as **'u'** (for up), the truck will move to the upward cell from the current one, i.e. (x,y-1) in the next turn.
    - If the instruction is set as **'s'** (for stay), the truck will stay on the current one, i.e. (x,y) in the next turn.
- Every element i of `pickUpCommands` and `dropOffCommands` must be set to a valid package id for each truck which needs to be picked up or dropped off after the turn change is requested to the helper, or -1 if no action needs to be taken. The truck needs to already be present at the cell before the turn starts for it to drop or pick up the package from that cell.
- All the trucks will start at (0,0) on the first turn, thus `truckPositions` will be set to [0,0] when the program first runs. The value of the elements can range from [0,0] to [N-1,N-1] for each truck.
- The `truckTurnsInTolls` array holds the number of turns for which each truck can no longer move due to being in a toll booth cell, or 0 otherwise. This will also be updated by the helper after each turn.

You can connect to the shared memory using the following:

```
MainSharedMemory *mainShmPtr;
shmget(key_t key, size_t sizeof(MainSharedMemory), int shmflg);
mainShmPtr = shmat(shmId, NULL, 0);
```

The `shmKey` will be provided to you in the input as described earlier.

The Main Helper Process

At every turn, including the first, the student program will be informed of the updates in the shared memory (turn change) by the helper process through a message of the following form:

```
// Helper to Student: Signals new turn state is ready in SHM
typedef struct TurnChangeResponse {
    long mtype;              <- fixed as 2
    int turnNumber;          <- the current turn (starts as 1)
    int newPackageRequestCount; <- count of new packages that have arrived this turn
    int errorOccured;        <- set as 1 if error
    int finished;            <- set as 1 if all requests have been fulfilled
} TurnChangeResponse;
```

The `mtype` of this will always be set to 2.

If `finished` has been set to 1, all the requests have been fulfilled by your program and the helper will calculate the metrics once your program has exited.

If `errorOccured` has been set to 1, your program has made an erroneous move, and the helper will print an error and exit once the turn change has been sent.

The `newPackageRequestCount` holds the count of new requests that have arrived in the current turn. If `newPackageRequestCount` has been set to K, the first K elements (0 to K-1) in the shared memory's `newPackageRequests` set have been set to the new requests.

Before the next turn, you are required to guess an authString for each of trucks that need to move in the subsequent turn. After you have set all the authStrings and commands for the trucks in the shared memory, you must send the following message to the helper requesting for a turn change:

```
// Student -> Helper: Signals turn commands are set in SHM
typedef struct TurnReadyRequest {
    long mtype;              <- fixed as 1
} TurnReadyRequest;
```

The `mtype` has to always be set as 1.
After receiving the message from your program, the helper will verify the authStrings and carry out all the commands as mentioned in the shared memory.
If the helper finds any of the authStrings to be incorrect, the helper will set errorOccured as 1 and print out an error and exit.
If any of the commands given in the shared memory are invalid, for example dropping off a package which is not picked up yet or not on the truck mentioned, the helper will perform the same routine and exit. If no error occurs, the helper will pickup and drop all the packages as instructed and carry out the movements for the respective trucks as mentioned in the shared memory. The helper will first process the pickups and drops and only then will the movements for that particular turn be carried out.

If a package is dropped at the cell in which it had initially requested to be dropped off at, the request will be considered completed. If it has been dropped off at any other cell, it needs to be picked up again and dropped at the original cell in the future turns for the request to be completed.
On any turn, only one package can be picked up for a particular truck. If multiple packages are present at the same cell and the same truck wants to pick them up, it can only be done in the subsequent turns. Similarly, only one package can be dropped off by a truck in any particular turn.
A package can only be dropped to a cell by a truck, but cannot be picked up again by another truck in the same turn. A truck can only carry a max of 20 packages, any instructions to pickup packages beyond this will result in an error.

The Authorization Strings and Solvers
You are provided with S independent solvers which can be used to guess the authorization strings. The key for the message queue of each solver is provided as part of the input.
You may communicate with the helpers using the following message:

```
// Student to Solver
typedef struct SolverRequest {
    long mtype;
    int truckNumber;
    char authStringGuess[TRUCK_MAX_CAP + 1];
} SolverRequest;
```

and the solver will reply with the following:

```c
// Solver to Student
typedef struct SolverResponse {
    long mtype; // set to 4
    int guessIsCorrect; // 1 if correct, 0 if incorrect
} SolverResponse;
```

Your program will have access to S solvers from the very first turn and the solvers will be ready to accept the guesses.
Each solver process can accept messages of two types: setting the truck number for which the guesses are being made (mtype=2), and guessing the actual authorization strings (mtype=3).

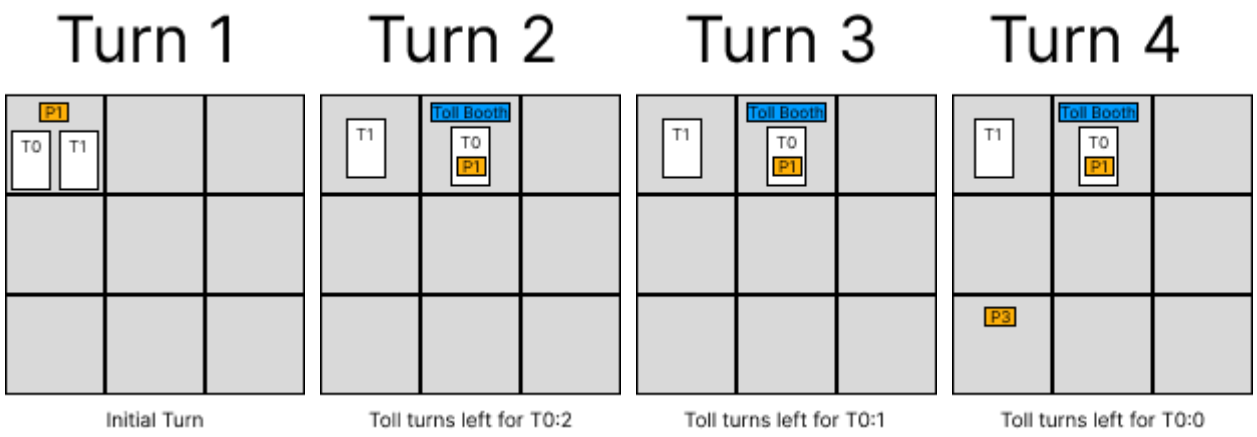
Thus, it is required to tell the solver for which truck the guesses are being made for with a message of **mtype 2.** In this case, the solver will set its target truck to the given value (between 0 and D-1), ignore the value sent for the authStringGuess variable, and won't send any reply back to your program.
Once the target truck has been set, your program can send messages of **mtype 3** with the authorization string guesses set in `authStringGuess` field, in which case, the solver will ignore the `truckNumber` field and reply back with a message of **mtype 4** telling you if the guess is correct or not.

Before the truck can carry out the movement instructions, you are required to input the correct sequence of moves in the form of an authorization string which has as many characters as there are packages in the truck (let that be L).
Each move in the sequence is one of four directions: Up, Down, Left, or Right. The solver process knows the secret combination.
You will send your guess as a string of L characters, using 'u' for Up, 'd' for Down, 'l' for Left, and 'r' for Right. The solver will check your combination and reply as stated above.
The number of packages considered will be the count of packages at the beginning of the turn. Any package that has been instructed to be picked up this turn will not be considered for the length of the authorization string. Similarly, if the truck is asked to drop off a package this turn, this package will still count towards the authorization string length.

You do not need to guess or set any authorization string for a truck if it is carrying zero packages, or if you do not wish to move it this turn.

A Step-by-Step Example

Initially:
There are 2 trucks available, and the grid is a 3×3 matrix.



Turn 1:
At the first turn, the helper function provides the initial state, where both trucks are positioned at (0,0) on the 3×3 grid.

In the helper's message on the message queue, the **newPackageRequestCount** variable has a value of 1. Therefore, the first element of the **newPackageRequests** array in shared memory will show this request — starting at (0,0) and requesting delivery to (2,2), with an **expiry_turn** of 8. Let's assume its request ID is 1.

Now, suppose Truck 0 has been assigned to deliver Package 1. Since it is already at (0,0), it will pick up the package. Next, if we want the truck to move right, the movement commands in the **truckMovementInstructions** array in shared memory will be **'d'** and **'s'**. As there are only two trucks, all elements of the array beyond the second element will be ignored.

To actually pick up the package, we set the first element of the **pickUpCommands** array to 1, and the remaining elements to -1, indicating that the package request with ID 1 is to be picked up by Truck 0.

Note that the truck starts this turn empty, so we do not need to provide any authorization string to move it.

Turn 2:
The turn begins with Truck 0 at (0,1) and Truck 1 at (0,0). No new package requests have appeared, so the helper's message will have **newPackageRequestCount = 0**.

Grid space (0,1) contains a toll booth. This is reflected in the **truckTurnsInToll** array in shared memory — its first element changes from 0 to 2, indicating that Truck 0 has entered a toll booth cell and must stay idle for the next two turns.

Therefore, the value of **truckMovementInstructions** for Truck 0 will not be considered for the next two turns.

Turn 3:
No new package requests have appeared. The **truckTurnsInToll** value for Truck 0 has now decreased to 1, while Truck 1 remains at (0,0).

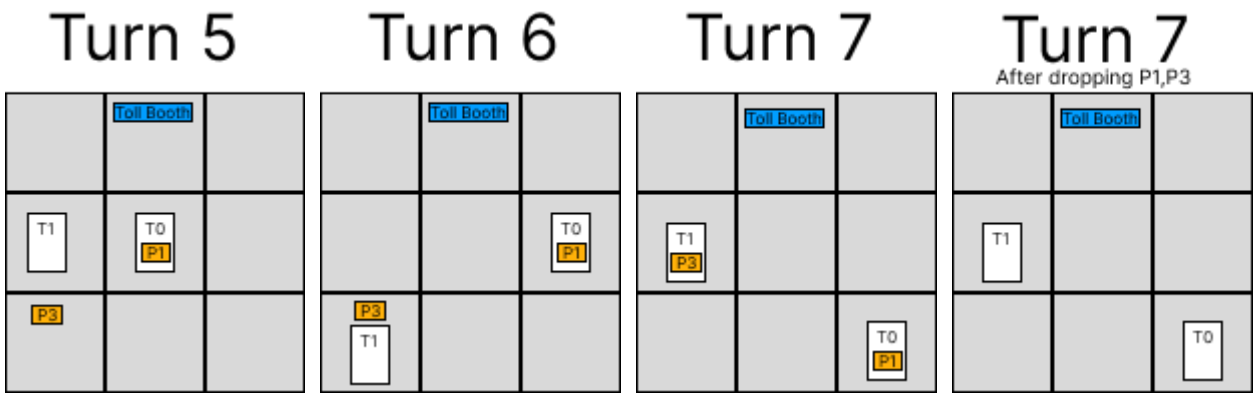
Turn 4:
The turn begins with Truck 0 at (0,1) and Truck 1 at (0,0). One new package request has appeared, so the helper's message will have **newPackageRequestCount = 1**.

This new package request, with ID 3, is to be picked up from (2,0) and delivered to (2,1), with an **expiry_turn** of 6. Truck 1 has been assigned to deliver the package with ID 3.

The **truckTurnsInToll** value for Truck 0 has now become 0, so it can move again. Since Truck 0 is not empty at the start of this turn and intends to move, it must find an authorization string of length 1, which, in this example, was **'d'**.

To find this string, we would select an available solver, send a message setting its target truck to Truck 0, and then try different string guesses until we find the correct one. Once found, we set **authStrings[0] = "d"**.

Now, both trucks move one square down.



Turn 5:
The turn begins with Truck 0 at (1,1) and Truck 1 at (1,0). Truck 0 wants to move right, while Truck 1 wants to move down. We need to find the authorization string only for Truck 0, as Truck 1 is not carrying any packages.

Turn 6:
The turn begins with Truck 0 at (1,2) and Truck 1 at (2,0). Truck 1 now picks up the package with ID 3, while Truck 0 needs to move down and Truck 1 needs to move up.

Truck 1 does not need an authorization string because it started the turn empty.

Turn 7:
The turn begins with Truck 0 at (2,2) and Truck 1 at (1,0). Both trucks now deliver their respective packages. However, the package with ID 3 expires, as its **expiry_turn** was 6 and it was delivered on turn 7. The package with ID 1, however, is successfully delivered.

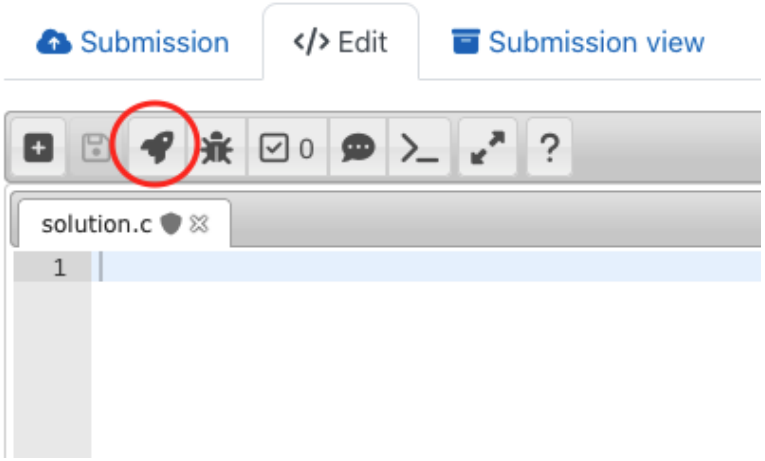
Evaluation:

- Only the last submission you make will be used for the grading. You can make as many submissions as you need to and test them against the public test cases.
- Part of the grade will be based on how many public and hidden test cases your submission passes.
- Part of your grade will depend on the number of turns your submission takes to solve a given test case. The lesser the number of turns, the more credit you get.
- Part of your grade will depend on how many expired packages you have delivered in total for a given test case. The lesser the number of expired packages, the more credit you get.
- The number of turns and the total number of expired packages will have equal weights in the grade.

Submitting and Testing your solution

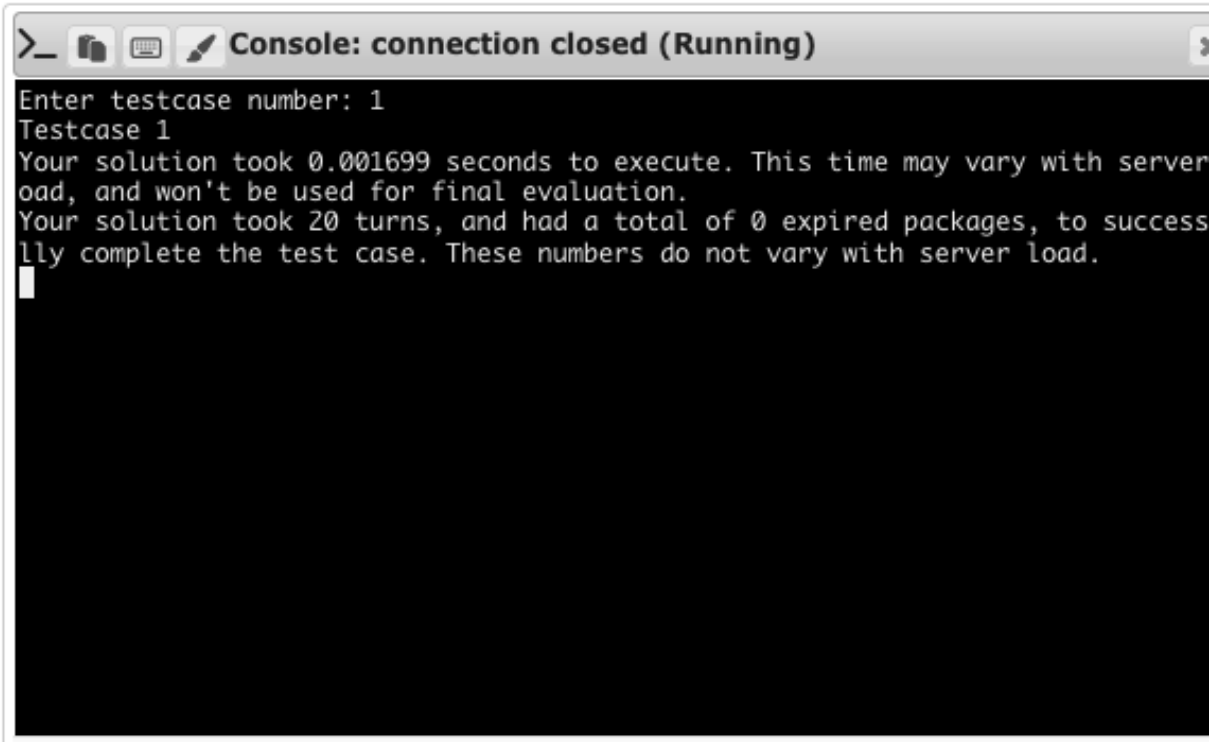
- The assignment is to be submitted on the portal:
  http://responsible-tech.bits- hyderabad.ac.in/moodle/
    - To login, Your username can be derived from your ID number thus:
        - If you ID Number is 2023A7PS1234H, then your username is 20231234

        - If this is your first login, you can get your password by clicking on the **Forgot Password** button and entering your **username**.
        - **Change your password after the first login to prevent potential tampering or theft of your code.**
    - On the RTMoodle portal, navigate to Operating Systems Autumn 2025 -> Assignment 2
- Assignment submission is available from outside the campus, ***but testing is only available from within the campus.***
- In the assignment, you can submit your code in the Submission tab, or directly write/paste it into the editor in the Edit tab.
- To test your code against the available test cases, you can click the Run button, as shown in the image below, on the Edit tab. The helper process and the other files will already be present and running on the platform, ready to communicate with your process.



After clicking the Run button, a terminal window will pop up with your code's output. Before running, though, it will first ask you to input which test case you wish to test your code against. Currently, there are four test cases, so you can enter a number from 1 to 4.

Important: The time limit for execution on VPL counts down even while it waits for you to enter which test case you wish to run, so enter it quickly.



- The platform will evaluate your program and show you the test cases which pass or fail and the time required for each test case. Please note that this time may vary based on how many students are simultaneously submitting on the server. Currently there is a time limit of 120 seconds for each test case and solutions to all the test cases must run within 120 seconds.
- There is also an 'evaluate' button that attempts to run all the test cases within the four second time limit. *Please ignore this button*, as even faster code may time out with it. Moreover, the final grading will be done later as it will involve hidden test cases and efficiency parameters.
- You may print values to stdout or stderr for testing purposes while writing your code.
- ***Guidelines for Cleanup:*** *Only detach from the shared memory in your process, do not use shmctl or msgctl to delete the shared memory and message queue.*