Problem Statement

You will be given a building with **K** floors and **N** elevators, and in every ‘turn’ you may move each elevator **at most one floor** up or down (or keep it where it is). On set turns, passengers will appear on specific floors, requesting to be sent to another floor. Your task is to write a POSIX compliant C program that controls these elevators so as to finish all the passengers’ requests in *as few turns as possible.*

Catch: On each turn, in order to move a non-empty elevator you must find a random **authorization string** whose length is equal to the number of people in the elevator. No string is required in case the elevator is empty, or if you don’t wish to move the elevator this turn.
The authorization string corresponding to a particular elevator can be obtained by communicating with one of the **solver processes** using a **message queue. There** will be **M** such solver processes available (**M <= N**), with a separate message queue for each.

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 elevator to move (1 floor up, 1 floor down, or no movement), as well as which passengers you would like to get picked up/dropped by which elevator (these passengers will be moved to/from the floor the elevator is currently on, before the elevator moves up or down 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.

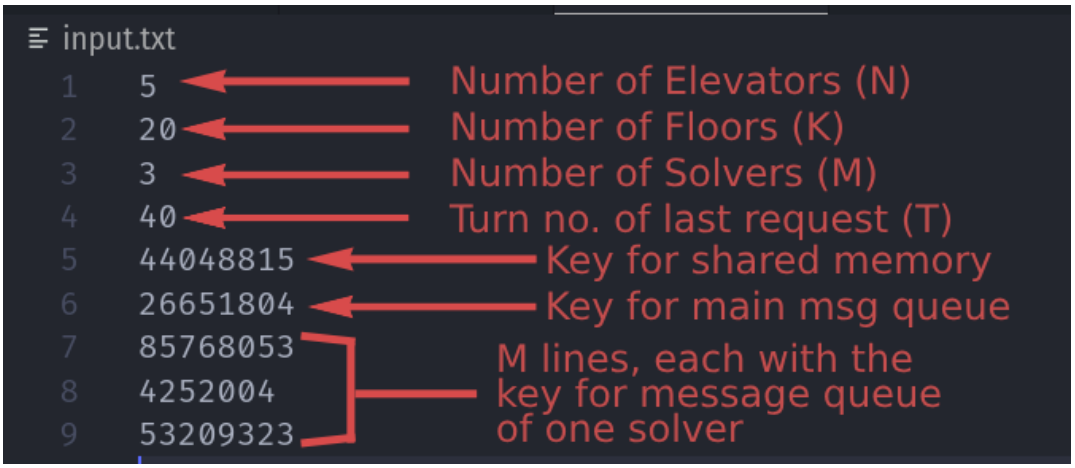
Once the strings have been verified, the helper will reply with the state of the new turn, allowing the process to be repeated.

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 won't 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 <= 100

K <= 500

M <= N

T <= 100

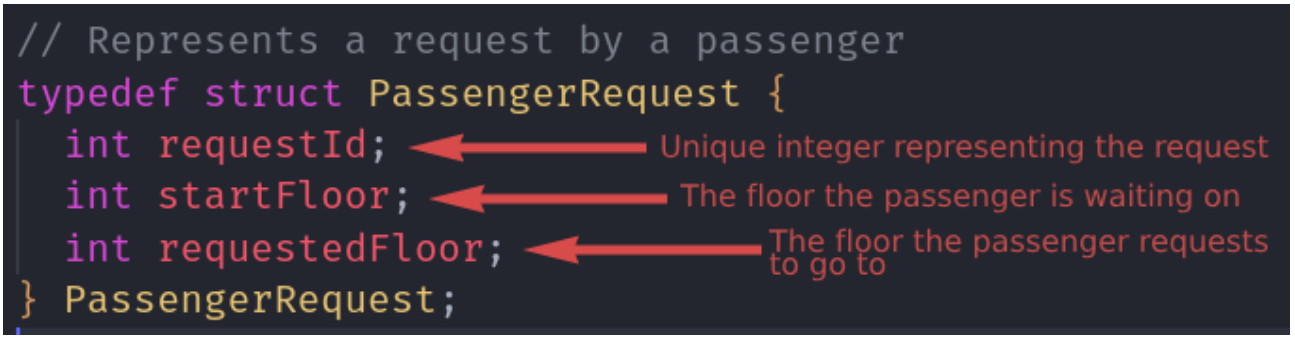
Each turn, no more than 30 new passenger requests may arrive.

Each authorization string will consist only of the first 6 letters of the alphabet, all lowercase.

Each elevator may hold no more than 20 passengers at a time.

The Requests

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



The Shared Memory

The shared memory is also represented as a struct, storing various values needed for your program and the helper to communicate:

```
// The shared memory shared between helper and student program
typedef struct MainSharedMemory {
  char authStrings[100][ELEVATOR_MAX_CAP + 1];          <- ith element is the auth string for elevator i
  char elevatorMovementInstructions[100];               <- ith element tells elevator i where to move
  PassengerRequest newPassengerRequests[MAX_NEW_REQUESTS];  <- New requests this turn
  int elevatorFloors[100];                              <- Floor each elevator is on at turn start
  int droppedPassengers[1000];                          <- Request IDs for each passenger to be dropped this turn
  int pickedUpPassengers[1000][2];                      <- Request IDs and elevator numbers for passengers to be picked up this turn
} MainSharedMemory;
```

Here, **MAX_NEW_REQUESTS** may be replaced by **30**, and **ELEVATOR_MAX_CAP** by **20**.
Some additional details on these values:

- `authStrings`, `elevatorMovementInstructions`, `droppedPassengers`, and `pickedUpPassengers` are to be set by your program every turn before asking the helper to proceed to the next turn. They don't need to be set before the first turn.
- `newPassengerRequests` and `elevatorFloors` will be set by the helper every turn before sending the state of the turn through the message queue. The student program may not change these.
- In `authStrings`, `elevatorMovementInstructions`, and `elevatorFloors` only the first N elements of the array (indices 0 to N-1) will be used, where N is the number of elevators.
- `elevatorMovementInstructions` accepts one of three values in each cell: **'u'** to move the elevator up by one floor, **'d'** to move it down by one floor, and **'s'** to make it stay on the same floor. The first two will require you to set the authString for that elevator, if the elevator isn't empty.
- Each element of `droppedPassengers` will be the integer request ID of one passenger request, corresponding to the passenger you want to drop from their elevator.
- Each element of `pickedUpPassengers` will be an integer array of length 2. In this array, the first element will be the request ID of the passenger you wish to get picked up by an elevator, and the second will be the elevator number (0 to N-1) corresponding to the elevator you want to send them to.
- Each elevator will start on the bottom floor. Thus, on turn 1, `elevatorFloors` will have value '0' in each element. In general, the value of each element will range from 0 to K-1.

Connect to the shared memory using the following statements:

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

Here, **shmKey** is the key for the shared memory taken as input earlier.

The Solver Processes and Authorization Strings

Each solver process is associated with one message queue (separate from the main queue used to communicate with the helper). The key for the message queue corresponding to each solver is part of the input. Every solver expects messages on its message queue of the type:

```
// These are the messages sent by the student program to the solvers
typedef struct SolverRequest {
  long mtype;
  int elevatorNumber;
  char authStringGuess[ELEVATOR_MAX_CAP + 1];
} SolverRequest;
```

and replies with messages of the type:

```
// These are the messages the solver responds with when a guess is made
typedef struct SolverResponse {
  long mtype;
  int guessIsCorrect;  // Is nonzero if the guess was correct
} SolverResponse;
```

There will be M such processes available. Each solver process will accept messages of one of two types: setting the target elevator (mtype = 2), and guessing the authorization string (mtype = 3). Thus, you must first tell the solver which elevator you wish to solve for, with a message of mtype 2. In this case, the solver will set its target elevator to the given value (between 0 and N-1), *ignore* the value sent for the `authStringGuess` variable, and *won't send any reply back to your program.*

Once the target elevator has been set, you may send messages of mtype 3. For each of these messages, the solver will *ignore* the value sent for `elevatorNumber` and reply back telling you whether your guessed string was the correct authorization string for the target elevator. The reply will be of **mtype 4**.

The correct authorization string for an elevator will have as many characters as there are people in that elevator. Each character of this string will be one of the first **6** letters of the alphabet, all lowercase. Note that the number of people in the elevator to be considered for the authorization string is the number of people at the beginning of the current turn. So, if the elevator is instructed to pick up people this turn, these new people **will not** count towards the number of characters in the string. Similarly, if the elevator is asked to drop off people this turn, these people **will** still count towards the authorization string length.

You do not need to guess or set any authorization string for an elevator if it is empty, or if you do not wish to move it this turn.

The Main Helper Process

Any turn, including the very first, will start with you receiving the current state of the elevators from the Helper, in a message of the following form:

```
// These are the messages the helper responds with when a new turn starts
typedef struct TurnChangeResponse {
  long mtype;                    <-- always 2
  int turnNumber;                <-- the current turn number (starts at 1)
  int newPassengerRequestCount;  <-- the number of new requests this turn
  int errorOccured;              <-- if 1, an error has occurred
  int finished;                  <-- if 1, the testcase is done
} TurnChangeResponse;
```

If `errorOccured` is 1, the helper will immediately print an error and attempt to exit after having sent the turn state.

If `finished` is 1, the helper will wait for your program to exit so it can calculate your statistics and perform cleanup.

The value of `newPassengerRequestCount` is the number of elements of the `newPassengerRequests` array (in the shared memory) which contain valid new requests this turn. For instance, if this value is 3, then the first three elements of `newPassengerRequests` (indices 0 to 2) contain new requests.

After setting all the authorization strings and commands for the elevators in the shared memory, you may send a message of the following form to the main helper:

```
// These are the messages sent by the student program to the helper
typedef struct TurnChangeRequest {
  long mtype;                    <-- must be 1
  int droppedPassengersCount;    <-- number of passengers dropped from their elevator this turn
  int pickedUpPassengersCount;   <-- number of passengers picked up by an elevator this turn
} TurnChangeRequest;
```

Here, `droppedPassengersCount` must equal the number of passengers you wish to drop off from their elevator this turn. For instance, if this value is 4, the helper will attempt to drop off passengers with requests IDs corresponding to the first four values of the `droppedPassengers` array (indices 0 to 3) in the shared memory.

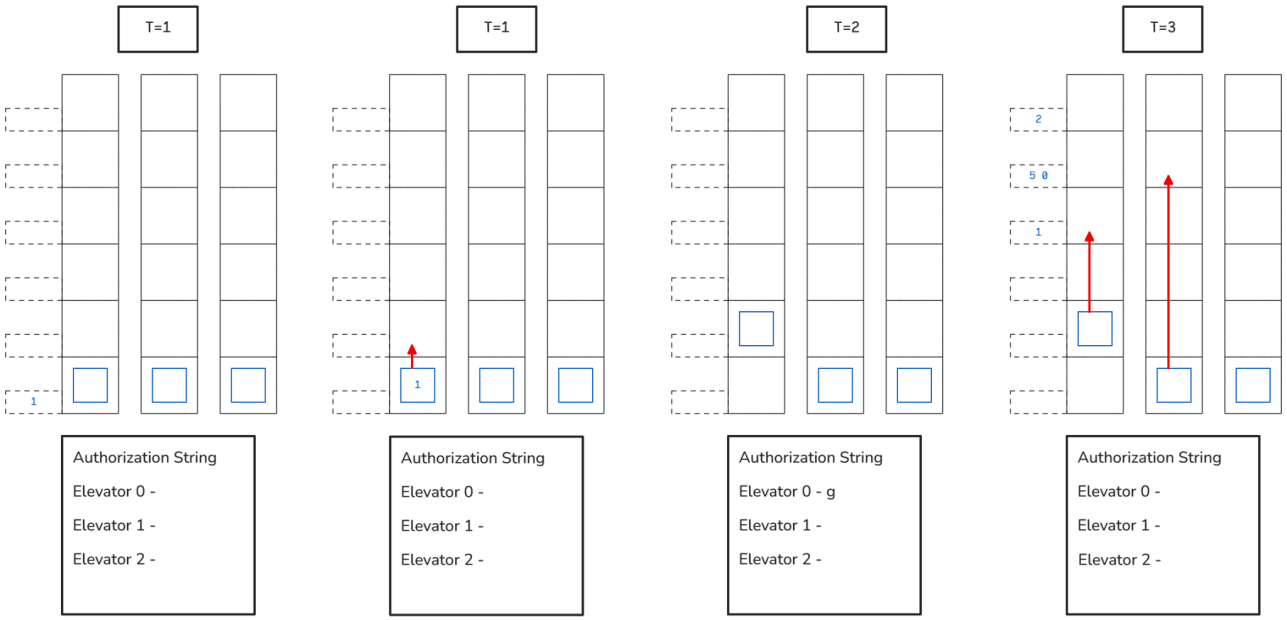
Similarly, `pickedUpPassengersCount` must equal the number of passengers to be picked up by some elevator, and the helper will act on elements of the `pickedUpPassengers` array in the shared memory according to this value.

After receiving your message, the helper will verify all the authorization strings for non-empty elevators that are instructed to move. If any of them is incorrect, the helper will print out the error, send a message with `errorOccured` set to 1, and exit. A similar situation will occur in case an

impossible command is given (dropping off a passenger who isn't in an elevator, trying to pick up a passenger with an elevator on a different floor, etc.). Otherwise, the elevators will pick up/drop any passengers as instructed, and then move up/down as instructed. The new state will be returned in the same format as given above.

If a passenger is dropped on the floor they originally requested to go to, their request will be considered finished. Meanwhile, if you drop them on a different floor, you will have to ensure they are picked up again from this floor and sent to their requested floor later.
On one turn, a passenger may only either enter or exit an elevator; they can't exit one elevator and enter another on the same turn, even if both elevators are on the same floor. An elevator, however, can have as many people enter/exit as needed in one turn, as long as the *final* count of people in the elevator does not exceed the maximum load limit of 20.

A step by step example



Consider the diagram shown above.

Turn 1
At the first turn, the helper gives you the initial state: all the elevators start on the bottom floor (floor 0). In this example, we also have a passenger on the bottom floor, who requests to go to floor 1.

In the helper's message on the message queue, we'd see the `newPassengerRequestCount` variable have a value of 1. Thus, the first element of the `newPassengerRequests` array in the shared memory will show us this request, starting at floor 0 and requesting to be taken to floor 1. Let's say its request ID is 4.

Suppose we want the first elevator (elevator 0) to pick this passenger up and move up a floor, while wanting the other elevators to stay where they are. Then, our movement commands in the

`elevatorMovementInstructions` array in the shared memory will be ‘u’, ‘s’, and ‘s’. As there are only three elevators, all the elements of the array beyond the third element will be ignored.

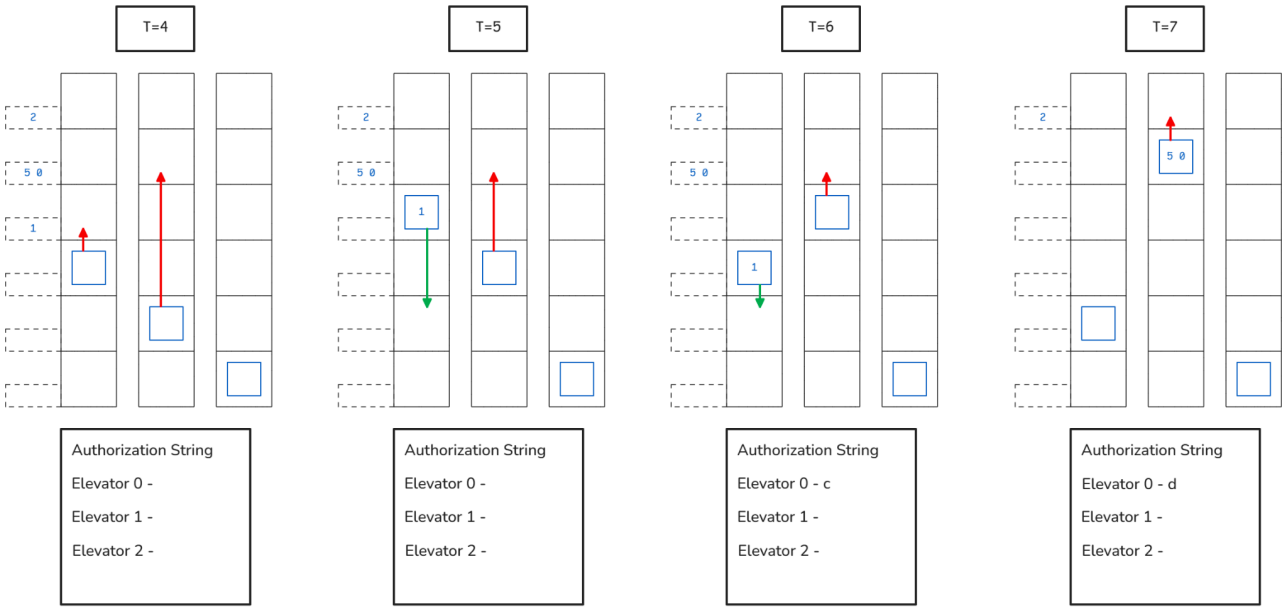
Meanwhile, to actually pick up this passenger, we’d make the first element of the `pickedUpPassengers` array be [4, 0], to represent that the passenger request with ID 4 is to be picked up by elevator 0. In our response to the helper on the message queue, we’d give `pickedUpPassengersCount` a value of 1.
Note that the elevator started this turn with being empty, and so we do not need to provide any auth string to move it up.

Turn 2
This turn begins with elevator 0 being on floor 1 with a passenger inside it, and the other two elevators being empty and at floor 0. No new passenger requests have shown up, thus our received message from the helper will have `newPassengerRequestCount` be 0.

We wish to drop off the passenger in elevator 0 as that elevator is at floor 1 already. For this, we’d put ‘4’ as the first element of the `droppedPassengers` array in the shared memory, and set `droppedPassengersCount` to ‘1’ in our message back to the helper. As there are no other requests, we can keep the elevators where they are, with movement instructions ‘s’, ‘s’, ‘s’.

Note that as elevator 0 started this turn with one passenger, if we had wanted to move this elevator we would have needed to guess a 1 character authorization string for it, which was ‘g’ in the above example. Guessing this would have required us to select an available solver, send a message setting its target elevator to elevator 0, then send messages with different string guesses until we got the correct string. In case we did this and moved the elevator, it would have first dropped off the passenger on floor 1 before moving.

Turn 3
The turn begins with our elevators on floors 1, 0, and 0, all empty. Now we have 4 new passenger requests, and thus choose to move the first two elevators up. As no elevator has passengers yet, we can move them without providing authorization strings.



Turn 4

Elevator 0 is now on floor 2, and elevator 1 is on floor 1. No new requests have arrived.

Turn 5

Elevator 0 moves to floor 3. This turn, we can ask it to pick up the passenger on this floor with the `pickedUpPassengers` array in the shared memory, and the `pickedUpPassengersCount` variable in our message, as before. If this request has an ID of 11, we'd pass [11, 0] as the first element of the `pickedUpPassengers` array.

We can also ask this elevator to move down now. As the elevator started the turn with no passenger, no authorization string was needed.

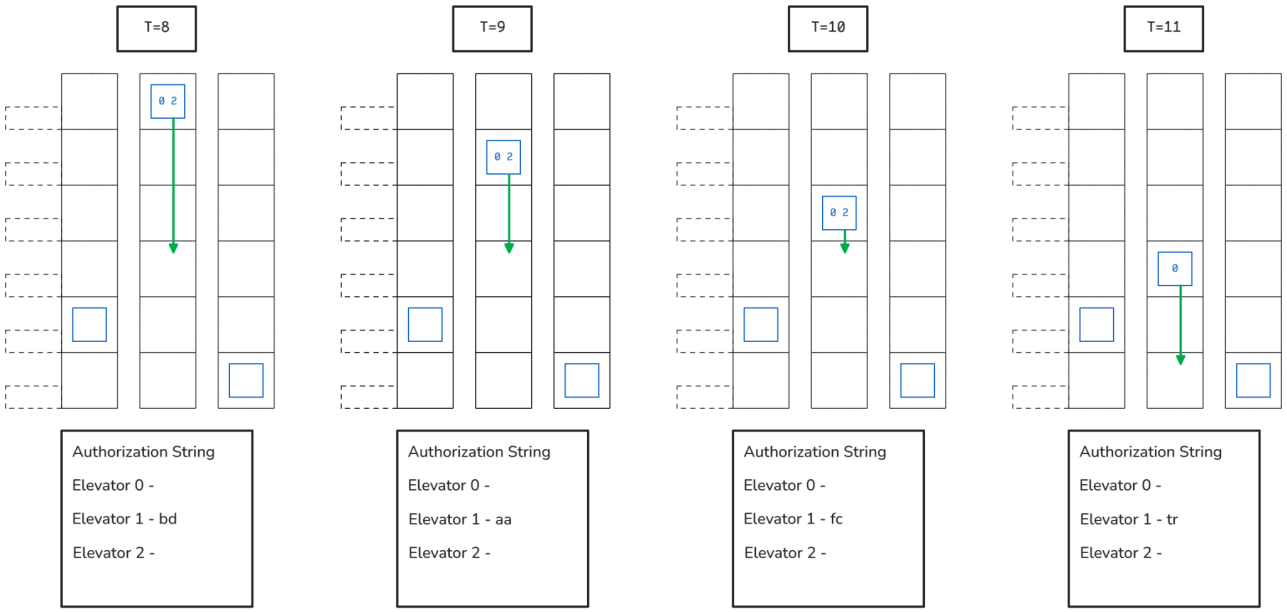
Turn 6

As elevator 0 has a passenger, continuing to move it down will require an authorization string. In this example, it is the string 'c'.

Turn 7

Elevator 0 starts on floor 1 with one passenger in it, who we want to drop off, which we can do as before.

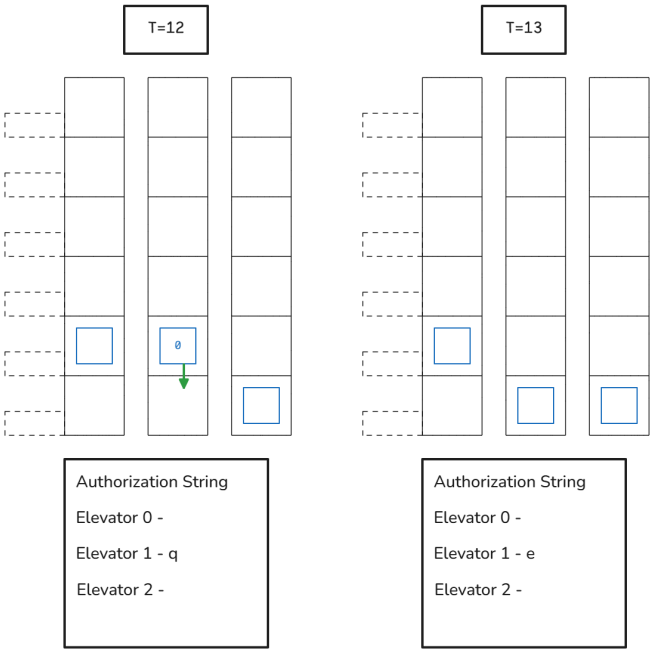
Meanwhile, Elevator 1 now has to pick up two passengers. If their request IDs are 3 and 7, we'd put two elements in the `pickedUpPassengers` array as [3, 1] and [7, 1] (the order won't make a difference). This turn, in the message we return to the helper, `pickedUpPassengersCount` has value 2, and `droppedPassengersCount` has value 1.



Turn 8

As elevator 1 had two passengers in it at turn start, moving it requires guessing an authorization string of length 2 using a solver.

The other turns continue similarly.



At turn 13, we ask elevator 1 to drop off its final passenger. If there are no more requests remaining in this test case, at turn 14 the helper will respond with a message having the `finished` variable set to '1'. At this point, we are done and must have our program exit.