Question 1 (20min)

Idea: iterate the matrix layer by layer, each layer consists of four sides in the following order: top, right, bottom and left. The last two sides are optional if the last layer has only two sides.
The image below illustrates the idea.

2	3	4	8
5	7	9	12
1	0	6	10

Top [colStart .. colEnd]
Right (rowStart …. rowEnd]
Bottom (colEnd .. colStart)
Left [rowEnd … rowStart)

Time complexity: O(N) where N is the total number of elements in matrix.

Question 2 (15 min)

Idea: divide the input number by base while it's not equal '0', at each step append the reminder to the result. In the end, reverse the result. If reminder is larger than '9' then subtract 10 from it and add char code of 'a'.

Java implementation is different from C because it preserves the sign, i.e. binary representation of -123 is -1111011 not 1111011. C implementation only preserves sign for base 10

Time complexity: O(N) where N is the total number of digits in the given number

Question 3 (1hr)

Endpoints
In this example all endpoints belong to *Gamemode* service

HTTP Method	path	response	description	Example
POST	**/report/:region/:gamemode**	200 - success 400 - if region	Reports gamemode for	POST- report /us/deathmatch

	region - country code (ISO 3166) gamemode - none empty string	doesn’t exist or gamemode is an empty string 201 - new game mode was created	the given region	— 200

GET	**/trending/:region**
	region - country code (ISO 3166)
		200 - success
		400 - if region doesn’t exist
		content-type : json
		json schema:
		{
		gamemode : String
		region: String
		}
			Get the most popular gamemode for the region	GET - /trending/gamemode/us
				{
				“gamemode” : “deathmatch”,
				“region”: “us”
				}

Service layer

```
public interface GamemodeService {

    /**
     * Reports the game mode for the given region. Creates a new game mode if it doesn't exist.
     *
     * @param gamemode game mode
     * @param region   country code in ISO 3166
     */
    void report(String gamemode, String region);

    /**
     * Gets the most popular game mode for the given region.
     *
     * @param region country code in ISO 3166
     * @return the most popular game mode
     */
    String getMostPopular(String region);
}
```

Database

Since we need to handle multiple concurrent requests we can't use a single database because it will become a bottleneck. Instead we might want to use sharding where the sharding key would be a *region* (country code two letters). **Note** that we are designing the database layer for the game service only. Some shards will receive more requests than others, e.g. a shard for USA will receive more traffic than the one for Sweden so we need to setup our boxes accordingly.

In this question, we might want to ask ourselves if we want to get the most popular game modes of all time or for some period of time, i.e. time window. The answer to this question will affect how we need to design our database layer. For example, in the former case, if some game mode was the most popular in the last year but has not been used during the last month, should it still be considered the most popular? Let's look at both scenarios.

For the first scenario the database table might look as follows:

Name	Type	Description
gamemode	varchar(200)	Game mode
region	varchar(2)	Country code
count	int unsigned	Total number of times this gamemode has been reported for the given region

PRIMARY KEY (\`gamemode\`, \`region\`)

Please see http://sqlfiddle.com/#!9/feae59a/1

Field count has **int unsigned** type, i.e 4294967295

Pros:

- Easy to implement
- Consumes less memory, O(N) where N is the total number of (gamemode, region) unique pairs
- Read Query is fast

Cons:

- Write operations won't scale well. In a case of heavy contention on the same row will cause performance degradation
- "count" field might overflow. We can use big int but that's not a strategic solution
- No way to get the most popular game modes for some time period

The second approach would be to create a new record for each reported (gamemode, region) pair, i.e. we will do append instead of modify. Each row will have a timestamp. However, in this approach, the gamemode service instances should have consistent clocks to avoid inaccurate reports.

For the first scenario the database table might look as follows:

Name	Type	Description
id	int	Unique id
gamemode	varchar(200)	Game mode
region	char(2)	Country code
timestamp	bigint	Timestamp in millis

Please see https://www.db-fiddle.com/f/uenvrnq2X1GC2wVVjw7RtP/0

Also, we need a job that will be triggered on schedule and delete rows with a timestamp older than our time window.

Pros:

- Possibility to get the most popular game modes for some time period
- No overflow issue
- Write operations will perform faster than in the first solution. Generally append operation is faster than modify

Cons:

- Memory consumption if higher than in the first approach. We can use the following formula to estimate total data size in mb for the given window:
  widnowSizeInDays * maxRequestsPerDay * size(row)
  size(row) = 4 + 8 + (1 + len(gamemode)) + 4 =~ 27bytes, where len(gamemode) ~= 10
  widnowSize = 30 day
  maxRequstsPerDay =   1M
  Total = 30 * 1M * 27bytes = **810mb**
- Read operations will be slower than in the first solution because we need to scan up to the maximum number of records that are within the time window (can be improved using caching, see the next section).

Caching

Instead of querying the database whenever a client calls getMostPopular we might want to use some caching solution. Once or a few times per day we will update the cache with a fresh data. Note: we need to make sure that clients are able to get data while the cache is updated.