Question 1

Develop a REST API that shares details about music tracks for a music streaming platform. This API can receive, show, and query data like the track title, album, description, and play count. The work is judged based on how accurately and efficiently the application handles and processes this data.

Music track data is represented as a JSON object, each describing various properties of a music track. The following properties are included:

- *id:* The unique integer ID of the object (Long).
- *title:* The title of the track (String).
- *albumName:* The corresponding album name of the track (String).
- *releaseDate:* The release date of the track, formatted as YYYY-MM-DD (Date).
- *playCount:* An integer value denoting how many times a user has played this track (Integer).

Example of a music track data JSON object:

```
{
 "title": "Lost in Echoes",
 "albumName": "Echoes of the Unknown",
 "releaseDate": "2021-07-15",
 "playCount": 5000
}
```

The REST service must expose the endpoint /music/platform/v1/tracks, enabling the management of music track records as follows:

POST request to /music/platform/v1/tracks:

- Creates a new music track record.
- It expects a valid music track data object as its body payload, excluding the id property.
- The service assigns a unique long id to the added object.
- The response includes the created record with its unique id, and the response code is 201.

GET request to /music/platform/v1/tracks:

- Responds with a list of all music track records and a response code of 200.

DELETE request to /music/platform/v1/tracks/{trackId}:

- Deletes the record with the specified track id if it exists in the database.

GET request to /music/platform/v1/tracks/sorted:

- Provides a list of music track records sorted by title.
- The response code is 200.
- The system defaults to sorting by title in ascending order.

Ensure the project meets all test case criteria when running the provided rspec tests. It uses the H2 database by default. Start by implementing the POST request to music/platform/v1/tracks, as testing other methods requires the POST functionality to work correctly.

Example requests and responses

POST request to /music/platform/v1/tracks

Request body:

```
{
  "title":"Lost in Echoes",
  "albumName":"Echoes of the Unknown",
  "releaseDate":"2021-07-15",
  "playCount":5000
}
```

The response code is 201, and when converted to JSON, the response body is:

```
{
  "id":1,
  "title":"Lost in Echoes",
  "albumName":"Echoes of the Unknown",
  "releaseDate":"2021-07-15",
  "playCount":5000
}
```

This adds a new object to the collection with the given properties and id 1.

GET request to /music/platform/v1/tracks

The response code is 200, and when converted to JSON, the response body, assuming these objects are in the collection, is as follows:

```
[
  {
    "id":1,
    "title":"Lost in Echoes",
    "albumName":"Echoes of the Unknown",
    "releaseDate":"2021-07-15",
    "playCount":5000
  },
  {
    "id":2,
```

```
    "title":"Bravos",
    "albumName":"Unknown",
    "releaseDate":"2021-07-16",
    "playCount":100
  }
]
```


DELETE request to /music/platform/v1/tracks/1
Assuming that the object with id 1 exists, the response code is 204, and the response body is empty.

GET request to /music/platform/v1/tracks/sorted
The response code is 200, and when converted to JSON, the response body, the returned collection sorted by its title ascending order, is as follows:


```
[
  {
    "id":1,
    "title":"Abacus",
    "albumName":"Echoes of the Unknown",
    "releaseDate":"2021-07-15",
    "playCount":5000
  },
  {
    "id":2,
    "title":"Bravos",
    "albumName":"Unknown",
    "releaseDate":"2021-07-16",
    "playCount":100
  }
]
```


Question 2

In the scope of Risk Management in global finance, you are tasked with implementing a class named *exchangeRate* to optimize investments. This class should include three methods with the same label but different arguments using method overloading.

Using method overloading, complete the implementation of the class *exchangeRate* with *3* methods.

1. *public String rate(double localCurrency, String foreignCurrency)* needs to process arguments like *rate(1, "Euro")* and should return *"1Euro".*
2. *public String rate(double localCurrency, double foreignCurrency)* must return the sum (as exchange rate approximation) when given inputs like *rate(1.1,2.2).* The resulting sum (like 3.3) should be rounded off to two decimal places.
    - Rounding Rules: Use the *BigDecimal* class with *RoundingMode.HALF_UP* for rounding. Follow these rounding rules:
        - 3.765 → 3.77
        - 3.7649 → 3.76
        - 3.778 → 3.79
    - Rules for returning the value: The value returned must be expressed using at least one decimal place. If the resulting sum after rounding off to two decimal places can be expressed using only a single decimal place, then the value with the single decimal place must be returned. For example:
        - 3.53 → 3.53
        - 3.00 → 3.0
        - 3.70 → 3.7
3. *public String rate(String localCurrency, String foreignCurrency)* should handle inputs like *rate("dollars","yen")* and return the concatenated string*: "dollarsyen".*

When code is submitted, the provided *Solution* class will test the add methods with different arguments.

Input Format For Custom Testing

The first line contains an integer, *n*, the number of inputs.

Each of the next *n* lines contains 2 space-separated values.

Sample Case 0

Sample Input For Custom Testing

```
2
4 1
 2 2
```

Sample Output

```
5.0
4.0
```

The resulting values will be computed as:

- Method 2: 4 + 1 = 5 = 5.00 (Rounded Off to 2 decimal places) = 5.0 (Must be expressed using a single decimal place)
- Method 2: 2 + 2 = 2 = 4.00 (Rounded Off to 2 decimal places) = 2.0 (Must be expressed using a single decimal place)

Question 3

In a URL shortening service, the URL is given as a string *url* consisting of lowercase English letters.

The task is to extract a subsequence of exactly *k* characters from the *url* that minimizes the URL compression, defined as the difference between the length of the subsequence and the number of distinct characters in the subsequence.

Implement a function to compute the minimum possible compression for any subsequence of length *k.*

The function *minimizeURLCompression* takes the following inputs:
string url: a string representing the original URL code
int k: the desired length of the subsequence

The function should return the minimum possible compression in any subsequence of length *k.*

Note: The subsequence of a string is a string obtained by deleting any set of characters from the original string while retaining the order of remaining characters.

Example
url = "ooxoxo"
k = 4

One optimal subsequence of length exactly *k* is "ooxx". The length of the subsequence is 4, and it contains two distinct characters, 'o' and 'x'. The URL compression is 4 - 2 = 2.

Hence, the answer is 2.

Constraints

- 2 ≤ *k* ≤ length(*url*) ≤ 105
- String *url* consists of lowercase English letters only.

Input Format For Custom Testing

The first line contains a string, *url*.
The next line contains an integer *k*.

Sample Case 0

Sample Input For Custom Testing

STDIN FUNCTION
----- --------
aaaabbbb → url = "aaaabbbb"
5 → k = 5

Sample Output

3

Explanation

One optimal subsequence of length exactly *k* is "aabbb". It is 5 characters long and has two distinct characters, 'a' and 'b'. The compression is 5 - 2 = 3.