BACKEND

Question 1:

Marks

15.00/60

Total Time

20 min 0 sec

Time Taken

11 min 48 sec

In the e-commerce industry, a company named "Ecom" has three tables to store information about its products: `products`, `categories`, and `suppliers`. The tables have the following structure:

```
**products**
| Column Name | Data Type |
| --- | --- |
| product_id | int |
| product_name | varchar(255) |
| category_id | int |
| supplier_id | int |
| price | decimal(10, 2) |

**categories**
| Column Name | Data Type |
| --- | --- |
| category_id | int |
| category_name | varchar(255) |

**suppliers**
| Column Name | Data Type |
| --- | --- |
| supplier_id | int |
| supplier_name | varchar(255) |
```

The company wants to develop a RESTful API to search for products based on their categories and suppliers. The API should be able to sort the results by price in ascending order.

Write a SQL query and a REST API implementation (using any language) to find the 3rd most expensive product in each category, considering only products that have a supplier. The API endpoint should accept a query parameter `category` to filter products by category.

PART 1: SQL Query

Goal: Find the **3rd most expensive product** in **each category**, considering **only products that have a supplier**.

✅ SQL Logic

We'll use the `DENSE_RANK()` window function to rank products by price **within each category**, in descending order (since we want the most expensive).

```
sql
CopyEdit
SELECT product_id, product_name, category_id, supplier_id, price
FROM (
    SELECT *,
           DENSE_RANK() OVER (PARTITION BY category_id ORDER BY price DESC)
AS rnk
    FROM products
    WHERE supplier_id IS NOT NULL
) ranked
WHERE rnk = 3;
```

✅ PART 2: REST API (OOP with Python Flask)

Let's now create a **RESTful API** in **Python using Flask** that:

- Accepts a **category ID** as a query parameter
- Returns the **3rd most expensive product** in that category with supplier info

---

✅ Flask API Code (OOP Style)

```
python
CopyEdit
from flask import Flask, request, jsonify
import mysql.connector

app = Flask(__name__)

class Database:
    def __init__(self):
        self.conn = mysql.connector.connect(
            host='localhost',
            user='your_user',
            password='your_password',
            database='ecom_db'
        )
        self.cursor = self.conn.cursor(dictionary=True)

    def get_3rd_expensive_product_by_category(self, category_id):
        query = """
        SELECT product_id, product_name, category_id, supplier_id, price
        FROM (
            SELECT *,
                   DENSE_RANK() OVER (PARTITION BY category_id ORDER BY
price DESC) AS rnk
            FROM products
            WHERE supplier_id IS NOT NULL
        ) ranked
        WHERE rnk = 3 AND category_id = %s;
        """
        self.cursor.execute(query, (category_id,))
        return self.cursor.fetchall()

db = Database()

@app.route('/products/third-expensive', methods=['GET'])
def get_third_expensive():
    category_id = request.args.get('category')
    if not category_id:
        return jsonify({"error": "Missing category parameter"}), 400

    results = db.get_3rd_expensive_product_by_category(category_id)
    return jsonify(results)

if __name__ == '__main__':
    app.run(debug=True)
```

Question 2:

Marks

15.00/60

Total Time

10 min 0 sec

Time Taken

10 min 0 sec

FloBiz is planning to develop a new feature in myBillBook that allows small and medium-sized businesses to generate invoices in different languages. The system should be able to handle multiple languages and format the invoices accordingly. For example, the system should be able to convert the date format from MM/DD/YYYY to DD/MM/YYYY for Hindi and Gujarati languages.

Design a class structure in OOPs that can handle the language-specific formatting for invoices. The class should be able to take in the language as an input and generate the formatted invoice accordingly. You can use any design pattern to solve this problem.

Assume that the system already has a class called `Invoice` that contains the necessary attributes for generating an invoice. You need to design a new class that can handle the language-specific formatting and integrate it with the existing `Invoice` class.
Write a PSEUDO_CODE for the new class structure and explain how it can handle the language-specific formatting for invoices.

Problem Summary:

We have an existing `Invoice` class. Now we want to format it differently depending on the selected language (like Hindi or Gujarati), e.g.:

- Change **date format** from `MM/DD/YYYY` to `DD/MM/YYYY`
- Support future additions like currency symbols, localized labels (like "Invoice", "Total")

✅ Goals:

1. Input language → apply correct format
2. Easily extensible (can add new languages without changing core logic)
3. Keep formatting logic separate from Invoice class

✅ Pseudo-Code Using Strategy Pattern

```
pseudo
CopyEdit
# Base Invoice Class (already exists)
class Invoice:
    date: string     # e.g., "07/21/2025"
    amount: float
    customer_name: string

    method print_invoice(formatter: InvoiceFormatter):
        formatted_invoice = formatter.format(self)
        return formatted_invoice

# Strategy Interface
class InvoiceFormatter:
    method format(invoice: Invoice): string
        pass
```

```
# English Formatter
class EnglishFormatter implements InvoiceFormatter:
    method format(invoice: Invoice):
        return "Invoice for " + invoice.customer_name +
               "\nDate: " + invoice.date +
               "\nAmount: $" + invoice.amount

# Hindi Formatter (MM/DD/YYYY → DD/MM/YYYY)
class HindiFormatter implements InvoiceFormatter:
    method format(invoice: Invoice):
        date_parts = split(invoice.date, "/")  # ["07", "21", "2025"]
        formatted_date = date_parts[1] + "/" + date_parts[0] + "/" +
date_parts[2]
        return "ग्राहक: " + invoice.customer_name +
               "\nदिनांक: " + formatted_date +
               "\nराशि: ₹" + invoice.amount

# Gujarati Formatter
class GujaratiFormatter implements InvoiceFormatter:
    method format(invoice: Invoice):
        date_parts = split(invoice.date, "/")
        formatted_date = date_parts[1] + "/" + date_parts[0] + "/" +
date_parts[2]
        return "ગ્રાહક: " + invoice.customer_name +
               "\nતારીખ: " + formatted_date +
               "\nરકમ: ₹" + invoice.amount

# Main Application Logic
method generate_invoice(language: string, invoice: Invoice):
    if language == "Hindi":
        formatter = HindiFormatter()
    else if language == "Gujarati":
        formatter = GujaratiFormatter()
    else:
        formatter = EnglishFormatter()

    return invoice.print_invoice(formatter)
```

✅ How This Works:

- The `generate_invoice()` function selects the right formatter class based on the language.
- Each formatter handles formatting (e.g., date style, labels) for its language.
- New languages = just add new formatter classes.

Question 3:

Marks

0.00/120

Total Time

20 min 0 sec

Time Taken

10 min 58 sec

Smallest window in a string containing all the characters of another string

Given two strings `s` and `t`. Find the smallest substring in the string `s` consisting of all the characters(including duplicates) of the string `t`.
Return `-1` in case there is no such substring present. In case there are multiple such windows of same length, return the one with the least starting index.
You have to complete `minWindow` function which consists of two strings `s` and `t` as inputs and return single string answer as output.

Input Format

First line contains string `s`.
Second line contains string `t`.

Output Format

Print an integer denoting the length of the smallest substring in `s` consisting of all the characters(including duplicates) of the string `t`.

Example 1

Input

```
timetopractice
toc
```

Output

```
toprac
```

Explanation

toprac is the smallest substring in which "toc" can be found.

Example 2

Input

```
zoomlazapzo
oza
```

Output

```
apzo
```

Explanation

apzo is the smallest substring in which "oza" can be found.

Constraints

```
1 <= |s|, |t| <= 10^5
```

Response:

```
#include <bits/stdc++.h>

using namespace std;


string minWindow(string s, string t) {

    if (t.size() > s.size()) return "-1";


    unordered_map<char, int> t_freq, window_freq;

    for (char c : t) t_freq[c]++;


    int start = 0, min_len = INT_MAX;

    int left = 0, right = 0;

    int formed = 0, required = t_freq.size();


    while (right < s.size()) {

        char c = s[right];

        window_freq[c]++;


        if (t_freq.count(c) && window_freq[c] == t_freq[c])

            formed++;
        // Try to shrink the window
        while (left <= right && formed == required) {
            if (right - left + 1 < min_len) {
                min_len = right - left + 1;
                start = left;
            }

            char l_char = s[left];
            window_freq[l_char]--;
            if (t_freq.count(l_char) && window_freq[l_char] < t_freq[l_char])
                formed--;
            left++;
        }
        right++;
    }

    return min_len == INT_MAX ? "-1" : s.substr(start, min_len);
}

int main() {
    string s, t;
    cin >> s >> t;
    cout << minWindow(s, t) << endl;
    return 0;
}
```

Shopping Options

You are given the list of costs of pants in an array “pants”, shirts in an array “shirts”, shoes in an array “shoes”, and skirts in an array “skirts”.

You are also given a budget amount ‘X’ to spend. You want to buy exactly 1 item from each list. Your task is to determine the total number of possible combinations that you can buy, given that the total amount of your purchase does not exceed your budget amount.

Input Format:

The first line of input contains an integer ‘T’ denoting the number of test cases to run. Then the test case follows.

- The first line of each test case contains five integers ‘P’, ‘Q’, ‘R’, ‘S’ and, ‘X’. Denotes the number of pants, shirts, shoes, skirts, and budget amount respectively.
- The second line of each test case contains exactly ‘P’ integers. Denotes the cost of each pant.
- The third line of each test case contains exactly ‘Q’ integers. Denotes the cost of each shirt.
- The fourth line of each test case contains exactly ‘R’ integers. Denotes the cost of each shoe.
- The fifth line of each test case contains exactly ‘S’ integers. Denotes the cost of each skirt.

Output Format :

For each test case, print an integer that represents the total number of combinations to buy that are valid according to the above conditions.

Output for each test case will be printed in a new line.

Example 1:

Input
```
2
2 2 1 1 10
3 4
2 4
2
```

```
1
3 2 2 1 15
3 4 6
1 4
2 3
8
```
Output :
```
3
3
```
Explanation :
In the first case,

There are a total of four ways to buy. Only valid purchases are [3, 2, 2, 1], [3, 4, 2, 1], [4, 2, 2, 1] because the sum of the cost of each item of each valid purchase is less than the budget amount.

In the second case,

There are a total of four ways to buy. Only valid purchases are [3, 1, 2, 8], [3, 1, 3, 8], [4, 1, 2, 8] because the sum of the cost of each item of each valid purchase is less than the budget amount.

Example 2

Input
```
2
3 3 3 3 14
3 4 7
1 3 5
6 8 10
1 2 3
4 3 2 2 20
2 8 7 10
5 5 5
6 3
4 5
```
Output :
```
12
21
```

Constraints:

1 <= T <= 100

1 <= ‘P’, ‘Q’, ‘R’, ‘S’ <= 10

1 <= ‘X’ <= 10^9

1 <= pants[i], shirts[i], shoes[i], skirts[i] <= 10^9

Time Limit: 1 sec

Response:

Final Complete Code (With Clean Structure):

```
cpp
CopyEdit
#include <bits/stdc++.h>
using namespace std;

// Utility function for binary search
int getSmallerNumbers(vector<int> &vec, int val) {
    int left = 0, right = vec.size();

    while (left < right) {
        int mid = (left + right) / 2;

        if (vec[mid] <= val) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return left;
}

// Class encapsulating the shopping logic
class ShoppingPlatform {
public:
    int shoppingOptions(vector<int>& pants, vector<int>& shirts,
vector<int>& shoes, vector<int>& skirts, int budget) {
        vector<int> pantShirtComb;

        // Store all pant + shirt combinations
        for (int p : pants) {
            for (int s : shirts) {
                pantShirtComb.push_back(p + s);
            }
        }

        vector<int> shoeSkirtComb;
        for (int sh : shoes) {
            for (int sk : skirts) {
                shoeSkirtComb.push_back(sh + sk);
            }
        }
        // Sort shoe + skirt sums for binary search
        sort(shoeSkirtComb.begin(), shoeSkirtComb.end());

        int totalWays = 0;

        for (int ps : pantShirtComb) {
            int rem = budget - ps;
            if (rem >= 0) {
                totalWays += getSmallerNumbers(shoeSkirtComb, rem);
            }
        }

        return totalWays;
    }
};

int main() {
    int t;
    cin >> t;

    while (t--) {
        int p, q, r, s, x;
        cin >> p >> q >> r >> s >> x;

        vector<int> pants(p), shirts(q), shoes(r), skirts(s);

        for (int i = 0; i < p; ++i) cin >> pants[i];
        for (int i = 0; i < q; ++i) cin >> shirts[i];
        for (int i = 0; i < r; ++i) cin >> shoes[i];
        for (int i = 0; i < s; ++i) cin >> skirts[i];

        ShoppingPlatform sp;
        cout << sp.shoppingOptions(pants, shirts, shoes, skirts, x) <<
endl;
    }

    return 0;
}
```

✅ Sample Input:

```
CopyEdit
2
2 2 1 1 10
3 4
2 4
2
1
3 2 2 1 15
3 4 6
1 4
2 3
8
```

✅ Sample Output:

```
CopyEdit
```

```
3
3
```

🔍 Key Concepts Used:

- `OOP`: Logic encapsulated inside a class `ShoppingPlatform`
- `Binary Search`: Via `getSmallerNumbers` to count valid combinations
- Efficient `O(N^2 log N)` solution using precomputation + search