Problem 1

Write a complete C++ program using stack to determine whether a given mathematical expression contains balanced brackets. The program should prompt the user to input an expression that may include different types of brackets such as parentheses (), curly braces {}, and square brackets []. Your program should use a stack to validate that the brackets are properly matched and correctly nested. If the brackets are balanced, display the message "Balanced"; otherwise, display "Not Balanced". In addition to the validation, the program must count and display the total number of brackets present in the expression and determine the maximum depth of nested brackets. As an added challenge, if the expression is not balanced, the program should also indicate the type and index position of the first unmatched bracket in the expression.

Problem 2

Design and implement a complete C++ program that simulates a customer service system using two queues. In this system, there are two types of customers: VIP and Regular. Use two separate queues to represent each category. The program should enqueue the following customers: VIP customers “Alice” and “Bob”, Regular customers “Charlie” and “Diana”, another VIP customer “Eve”, and Regular customers “Frank” and “Grace”. The customer service logic should always serve one VIP customer (if any) followed by one Regular customer (if any), and repeat this cycle until both queues are empty. As each customer is served, display their name and type. Once all customers have been served, print how many customers remain in each queue and indicate whether either queue is empty. For an added challenge, allow the user to enter new customers dynamically (including their name and type) until they type "exit" to stop the input process.

Problem 3

Write a complete C++ program using stack to simulate the **undo functionality of a simple text editor**. In this simulation, users perform a series of actions such as typing characters, deleting text, or applying formatting (like bold or italic). Each action should be recorded as a string and pushed onto a stack, representing the most recent operations. The user should be able to perform a sequence of actions and then undo a certain number of them. When the user performs an undo, the last action should be popped from the stack and displayed as the operation being undone. After all undo operations are completed, display the list of remaining actions still in effect (i.e., those that were not undone). Include the following features:

1. Accept a list of actions from the user.
2. Allow the user to specify how many undo operations to perform.
3. After each undo, show what action was reversed.
4. Display the final state of the action stack (remaining actions).

Problem 4

Write a C++ program using queue to simulate a print queue in an office environment. Each print job will have a unique ID, a document name, and a number of pages. The program should enqueue a list of print jobs (at least six), each with randomly assigned page counts. Simulate the printer processing each job in order of arrival, where each job takes a number of seconds proportional to its page count (assume 1 second per page). As the jobs are processed, print detailed information including the job ID, document name, pages, and estimated completion time. After all jobs are processed, display the average number of pages per job and the total processing time. For an added challenge, allow new jobs to arrive dynamically during processing and prioritize them if they have fewer than 3 pages.

Problem 5

Write a complete C++ program that simulates a **web browser's navigation system** using both a **stack** and a **queue**. In this simulation, the browser keeps track of visited websites, supports backward/forward navigation using stacks, and also maintains a queue for download tasks. The program should perform the following operations:

1. Simulate the user visiting a sequence of websites (e.g., "google.com", "youtube.com", "github.com", etc.). Each visited URL should be pushed onto a **history stack**.
2. Allow the user to press the **back button**, which pops the current page from the history stack and pushes it onto a **forward stack**, simulating back navigation.
3. Allow the user to press the **forward button**, which pops a page from the forward stack and pushes it back onto the history stack, simulating forward navigation.
4. Separately, simulate a **download queue** where files are queued for downloading. Add several download tasks (file names) to a queue. As the download proceeds, dequeue the files one by one and print their download status.
5. At the end of the simulation:
    - Display the current page being viewed.
    - Show the list of pages that can be navigated forward and backward.
    - Show the remaining files in the download queue (if any).