Week 6 - Live session

Question 1

Expression contains redundant bracket or not

Given a string of balanced expressions, find if it contains redundant parentheses or not. A set of parentheses are redundant if the same sub-expression is surrounded by unnecessary or multiple brackets. Print ‘Yes’ if redundant else ‘No’.
Note: Expression may contain ‘+‘, ‘*‘, ‘–‘ and ‘/‘ operators. Given expression is **valid** and there are **no white** spaces present.

Example:
Input:
((a+b))
(a+(b)/c)
(a+b*(c-d))
Output:
Yes
Yes
No

Explanation:
1. ((a+b)) can reduced to (a+b), this Redundant
2. (a+(b)/c) can reduced to (a+b/c) because **b** is surrounded by **()** which is redundant.
3. (a+b*(c-d)) doesn't have any redundant or multiple brackets.

```
import java.util.Stack;
public class GFG {
// Function to check redundant brackets in a
// balanced expression

    static boolean checkRedundancy(String s) {
        // create a stack of characters
        Stack<Character> st = new Stack<>();
    char[] str = s.toCharArray();
    // Iterate through the given expression
    for (char ch : str) {

        // if current character is close parenthesis ')'
        if (ch == ')') {
            char top = st.peek();
            st.pop();

            // If immediate pop have open parenthesis '('
            // duplicate brackets found
            boolean flag = true;

            while (top != '(') {

                // Check for operators in expression
                if (top == '+' || top == '-'
                        || top == '*' || top == '/') {
                    flag = false;
                }

                // Fetch top element of stack
                top = st.peek();
                st.pop();
            }

            // If operators not found
            if (flag == true) {
                return true;
            }
        } else {
            st.push(ch); // push open parenthesis '(',
        }                    // operators and operands to stack
    }
    return false;
}
```

Week 6 - Live session

```
// Function to check redundant brackets
    static void findRedundant(String str) {
        boolean ans = checkRedundancy(str);
        if (ans == true) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }

// Driver code
    public static void main(String[] args) {
        String str = "((a+b))";
        findRedundant(str);

        str = "(a+(b)/c)";
        findRedundant(str);

        str = "(a+b*(c-d))";
        findRedundant(str);
    }
}
```

Question 2

Problem

Round robin algorithm is a very popular scheduling algorithm. It allocates a quantum to process in a sequential manner to run. Every process has access to the CPU for the entire quantum.
Your task is to implement this algorithm using a queue.

Input Format

No of process
Burst time of processes
Quantum size

Output Format

Current sequence in queue

Example

Input:

```
4
4
3
2
2
2
```

Output:

```
4 3 2 2
3 2 2 2
2 2 2 1
2 2 1
2 1
1
```

Test Case 1

Input:

```
5
```

```
10
8
4
6
2
2
```

Output:

```
10 8 4 6 2
8 4 6 2 8
4 6 2 8 6
6 2 8 6 2
2 8 6 2 4
8 6 2 4
6 2 4 6
2 4 6 4
4 6 4
6 4 2
4 2 4
2 4 2
4 2
2 2
2
```

Test Case 2

Input:

```
6
4
10
1
7
9
4
4
```

Output:

```
4 10 1 7 9 4
10 1 7 9 4
1 7 9 4 6
7 9 4 6
9 4 6 3
4 6 3 5
6 3 5
3 5 2
5 2
2 1
1
```

Test Case 3

Input:

```
5
10
34
56
24
3
20
```

Output:

```
10 34 56 24 3
34 56 24 3
56 24 3 14
24 3 14 36
3 14 36 4
14 36 4
36 4
4 16
```

16

Solution

Solution

```
import java.util.Scanner;

class Queue{
        int front, rear;
        int capacity, size;
        int[] arr;
        Queue(int noOfProcess){
                capacity = noOfProcess;
                arr = new int[capacity];
                front = -1;
                rear = -1;
```

```
    }

    void enqueue(int ele){
        if(rear==-1)
            front=0;
        rear++;
        arr[rear] = ele;
        size++;

    }

    int dequeue(){
        int ele = arr[front];
        shift();
        size--;
        return ele;

    }

    void shift(){
        for(int i=front; i<rear;i++)
            arr[i] = arr[i+1];
        rear--;
    }

    boolean isEmpty(){
        if(size==0)
            return true;
        return false;
    }

    void display(){
        for(int i=front; i<=rear;i++)
            System.out.print(arr[i]+" ");
        System.out.println();
    }
}
```

```
class RoundRobinScheduler{



    static void schedular(int[] burstTime, int noOfProcess, int quantum){
           Queue queue = new Queue(noOfProcess);
           for(int i = 0; i < noOfProcess; i++)
                      queue.enqueue(burstTime[i]);
           while(!queue.isEmpty()){
                      queue.display();
                      int curr = queue.dequeue();
                      if(curr > quantum)
                                 queue.enqueue(curr - quantum);
           }

    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int noOfProcess = in.nextInt();
           int[] burstTime = new int[noOfProcess];
           for(int i=0; i<noOfProcess; i++)
                      burstTime[i] = in.nextInt();
           int quantum = in.nextInt();

        System.out.println(schedular(burstTime, noOfProcess, quantum));
    }
}
```