Felvételi, 2025. szeptember
Szoftverfejlesztés mesterképzés
Javítókulcs

Programozás és programozási nyelvek

```
int fuggveny ( int n, int x [][100] ){
      int maxOsszeg = x[0][0] + x[0][n-1];
      for ( int i = 0 ; i < n ; ++i ){
        maxOsszeg = max(maxOsszeg,x[0][i]+x[i][n-1]);
        maxOsszeg = max(maxOsszeg,x[i][n-1]+x[n-1][n-1-i]);
        maxOsszeg = max(maxOsszeg,x[n-1][n-1-i]+x[n-1-i][0]);
        maxOsszeg = max(maxOsszeg,x[n-1-i][0]+x[0][i]);
      }
      return maxOsszeg;
}
```

Adatbázisok

Adott egy egyetem részleges adatbázisa:

Student(sid, name, username, gpa)

Course(cid, name)

Enrolled(sid, cid, grade)

Írj Sql lekérdezéseket a következő feladatok megoldására:

a) Írasd ki azt a kurzust, amelyikre 20-al többen jelentkeztek mint az Adatbázisrendszerek kurzusra

```
SELECT  c.name
FROM    Course c JOIN  Enrolled e ON c.cid = e.cid
GROUP BY  c.name
HAVING COUNT(e.sid) >= (
        SELECT COUNT(e2.sid) +20
        FROM   Course c2  JOIN  Enrolled e2 ON c2.cid = e2.cid
```

```
        WHERE  c2.name = 'Adatbázisrendszerek' );
```

b) Keresd meg azt a legmagasabb GPA-val rendelkező hallgatót minden olyan kurzuson, ahol a diákok átlagos GPA-ja 3.0 alatt van, és a kurzusra legalább 5 hallgató jelentkezett.

```
WITH CTE_Course_Summary AS (
    SELECT e.cid, COUNT(e.sid), AVG(s.gpa)
    FROM Enrolled e
    JOIN Student s ON e.sid = s.sid
    GROUP BY  e.cid
    HAVING COUNT(e.sid) >= 5 AND AVG(s.gpa) < 3.0
),
CTE_Ranked_Students AS (
    SELECT  e.sid, e.cid,
            RANK() OVER (PARTITION BY e.cid ORDER BY s.gpa DESC) AS rnk
    FROM Enrolled e JOIN Student s ON e.sid = s.sid
)
SELECT
    c.name, s.name, s.gpa
FROM Course c
JOIN CTE_Course_Summary cs ON c.cid = cs.cid
JOIN CTE_Ranked_Students rs ON cs.cid = rs.cid
JOIN Student s ON rs.sid = s.sid
WHERE rs.rnk = 1;
```

c) Írasd ki a legnagyobb azonosítójú (Sid) hallgató adatait, aki legalább egy kurzusra beiratkozott.

```
SELECT sid, name
 FROM student
WHERE sid IN (   SELECT MAX(sid) FROM enrolled );
```

d) Keresd meg az összes olyan kurzust, amelyre nincsenek hallgatók beiratkozva.

```
SELECT * FROM course
 WHERE NOT EXISTS (
                SELECT * FROM enrolled    WHERE course.cid = enrolled.cid);
```

e) Számítsd ki az egyes kurzusokra beiratkozott hallgatók számát és az átlagos tanulmányi átlagukat (GPA). Rendezd a kimenetet a kurzusok beiratkozási száma szerint növekvő sorrendbe.

```
   SELECT c.name, AVG(s.gpa),  count(*)
   FROM enrolled  e JOIN student  s
   ON e.sid = s.sid
   join course c on c.cid =e.cid
   GROUP BY c.name;
```

VAGY

```
SELECT * FROM course c,
 LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
       WHERE enrolled.cid = c.cid) t1,
 LATERAL (SELECT AVG(gpa) avg FROM student s
          JOIN enrolled  e ON s.sid = e.sid
         WHERE e.cid = c.cid)  t2
```

```
 ORDER BY cnt ASC;
```

f) Keresd meg minden kurzus esetén azt a hallgatót, aki a második legmagasabb osztályzatot érte el.

```
SELECT c.name, s.name, e.grade
FROM  Student s JOIN   Enrolled e ON s.sid = e.sid
JOIN  Course c ON e.cid = c.cid
WHERE  (e.cid, e.grade) IN (
        SELECT  cid,  grade
        FROM (
            SELECT cid, grade, DENSE_RANK() OVER (PARTITION BY cid
ORDER BY grade DESC) AS dr_rank
            FROM  Enrolled )
        WHERE dr_rank = 2
    );
VAGY
SELECT c.name, t.name,  t.grade
FROM  Course c
CROSS JOIN  LATERAL (
        SELECT  s.name ,  e.grade
        FROM Enrolled e  JOIN Student s ON e.sid = s.sid
        WHERE e.cid = c.cid
        ORDER BY  e.grade DESC
        OFFSET 1 ROW
        FETCH NEXT 1 ROW ONLY
    ) t;
```

Objektumelvű programozás és szoftvertervezés

Elv: 0.5 pont, Jó példa: 0.75, Ellenpélda: 0.75

1. S – Single Responsibility Principle (SRP)

Egy osztálynak csak egyetlen felelőssége legyen.

☑ **Betartja:**

```
class InvoicePrinter {
    void print(Invoice invoice) {
        System.out.println("Printing invoice: " + invoice.getId());
    }
}

class InvoiceSaver {
    void save(Invoice invoice) {
        // save to DB
    }
}
```

❌ **Megszegi:**

```
class InvoiceManager {
    void print(Invoice invoice) { /* printing */ }
    void save(Invoice invoice) { /* saving */ }
}
```

(Egy osztály túl sok mindent csinál.)

---

2. O – Open/Closed Principle (OCP)

A kód legyen nyitott a bővítésre, de zárt a módosításra.

☑ **Betartja:**

```
interface Shape {
    double area();
}

class Circle implements Shape {
    private double r;
    public Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
}

class Rectangle implements Shape {
    private double w, h;
    public Rectangle(double w, double h) { this.w = w; this.h = h; }
    public double area() { return w * h; }
}
```

❌ **Megszegi:**

```
class AreaCalculator {
    double area(Object shape) {
        if (shape instanceof Circle c) {
            return Math.PI * c.r * c.r;
        } else if (shape instanceof Rectangle r) {
            return r.w * r.h;
        }
        return 0;
    }
}
```

(Új alakzatnál mindig módosítani kell a kódot.)

---

3. L – Liskov Substitution Principle (LSP)

Az alaposztály objektumait mindig helyettesíthetővé kell tenni a származtatott osztályok objektumaival anélkül, hogy a program helyessége megváltozna.

☑ **Betartja:**

```
abstract class Shape {
    public abstract double calculateArea();
}

class Rectangle extends Shape {
    protected double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

public class LiskovExample {
    // Helyes példa - minden Shape ugyanúgy viselkedik
    public static void testShapes() {
        Shape[] shapes = {
            new Rectangle(5, 4),
            new Circle(3)
        };

        for (Shape shape : shapes) {
            System.out.println("Terület: " + shape.calculateArea());
        }
```

```
    }
```

❌ **Megszegi:**

```
class BadRectangle {
    protected double width, height;

    public void setWidth(double width) { this.width = width; }
    public void setHeight(double height) { this.height = height; }
    public double getArea() { return width * height; }
}

// Square örökli BadRectangle-t, de megszegi annak viselkedését
class Square extends BadRectangle {
    @Override
    public void setWidth(double width) {
        this.width = width;
        this.height = width; // Négyzetnél mindkét oldal egyenlő
    }

    @Override
    public void setHeight(double height) {
        this.width = height;
        this.height = height; // Négyzetnél mindkét oldal egyenlő
    }
}
```

```
public class LiskovExample {
    // Rossz példa - a Square nem viselkedik úgy, mint a Rectangle
    public static void testRectangle(BadRectangle rect) {
        rect.setWidth(5);
        rect.setHeight(4);
        System.out.println("Elvárva: 20, Kapott: " + rect.getArea());
        // Rectangle esetén: 20, Square esetén: 16 (váratlan!)
    }
Ú
```

---

4. I – Interface Segregation Principle (ISP)

A klienseknek nem kell olyan interfészeket implementálniuk, amiket nem használnak. Ne készítsünk „kövér” interfészeket.

☑ **Betartja:**

```
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}

class MultiFunctionPrinter implements Printer, Scanner {
    public void print() { /*...*/ }
    public void scan() { /*...*/ }
}

class SimplePrinter implements Printer {
    public void print() { /*...*/ }
}
```

❌ **Megszegi:**

```
interface Machine {
    void print();
    void scan();
}

class OldPrinter implements Machine {
    public void print() { /*...*/ }
    public void scan() { throw new UnsupportedOperationException(); }
}
```

(`OldPrinter`-t feleslegesen kényszeríti a `scan()` implementálására.)

---

5. D – Dependency Inversion Principle (DIP)

Az absztrakciókra támaszkodjunk, ne a konkrét implementációkra.

☑ **Betartja:**

```
interface NotificationService {
    void send(String message);
}

class EmailService implements NotificationService {
    public void send(String message) { System.out.println("Email: " + message);
}
}

class OrderProcessor {
    private final NotificationService notifier;
    OrderProcessor(NotificationService notifier) {
        this.notifier = notifier;
    }
    void processOrder() {
        notifier.send("Order processed");
    }
}
```

❌ **Megszegi:**

```
class OrderProcessor {
    private final EmailService emailService = new EmailService();
    void processOrder() {
        emailService.send("Order processed");
    }
}
```

(`OrderProcessor` szorosan kötődik az `EmailService`-hez.)