# DEPARTMENT OF COMPUTER ENGINEERING

| | |
|---|---|
| Semester | T.E. Semester VI – Computer Engineering |
| Subject | Mobile Computing |
| Subject Professor In-charge | Prof. Sneha Annappanavar |
| Assisting Teachers | Prof. Sneha Annappanavar |
| Laboratory | M310A |

| | |
|---|---|
| Student Name | Deep Salunkhe |
| Roll Number | 21102A0014 |
| TE Division | A |

**Title:**

**Frequency Reuse**

---

**Explanation:**

1. **Importing Libraries**:
    - The code imports the **turtle**, **math**, and **random** libraries, which are used for graphics drawing and generating random colors.
2. **Global Variables**:
    - **size**: Represents the size of each hexagon.
    - **label**: Represents the label number for each hexagon.
    - **cluster**: Represents the cluster of hexagons.
3. **Helper Functions**:
    - **round_position(position)**: Rounds the position coordinates to two decimal places.
    - **draw_hexagon(drawer, cords)**: Draws a hexagon with the specified coordinates.
    - **color_hexagon(drawer, cords, color)**: Draws and fills a hexagon with the specified color.
    - **calculate_adjacent_centers(centers, visited, cords)**: Calculates the coordinates of adjacent hexagons.

- **check_input(cords, visited, input_clr)**: Checks if adjacent hexagons have the same color.

4. **Drawing the Cluster** (**draw_cluster(n)**):
    - Draws a cluster of hexagons based on the given input **n**.
    - Initializes a turtle drawer and sets its speed to maximum.
    - Iteratively draws hexagons and calculates adjacent centers until **n** hexagons are drawn.
    - Asks the user to input colors for each hexagon and checks if adjacent hexagons have the same color.
    - If the colors are invalid, prints a message.
5. **Input Colors** (**input_colors(n)**):
    - Asks the user to input colors for each hexagon.
    - Returns a list of input colors.
6. **Main Function**:
    - Prompts the user to enter values for **i** and **j**.
    - Calculates the number of hexagons (**n**) based on the input values.
    - Draws the cluster of hexagons and waits for user interaction.
7. **Random Color Function** (**random_color()**):
    - Generates a random RGB color.

Overall, this code uses the **turtle** library to draw a cluster of hexagons on the screen. It prompts the user to input colors for each hexagon and ensures that adjacent hexagons have different colors. This code can be used for visualizing and experimenting with hexagonal grids.

---

**Implementation:**

```
import turtle
import math
import random


size = 20
label = 1
cluster = []
```

```
def round_position(position):
    return round(position[0], 2), round(position[1], 2)


def draw_hexagon(drawer, cords):
    global size, label
    x, y = cords

    vertices = [
        (-size, 0),
        (-size / 2, -(3**0.5) / 2 * size),
        (size / 2, -(3**0.5) / 2 * size),
        (size, 0),
        (size / 2, (3**0.5) / 2 * size),
        (-size / 2, (3**0.5) / 2 * size),
    ]

    drawer.penup()
    drawer.goto(x, y - 8)
    drawer.write(label, align="center")

    drawer.goto(x + vertices[0][0], y + vertices[0][1])
    drawer.pendown()

    for vertex in vertices[1:]:
        drawer.goto(x + vertex[0], y + vertex[1])

    drawer.goto(x + vertices[0][0], y + vertices[0][1])


def color_hexagon(drawer, cords, color):
    global size, label
    x, y = cords

    vertices = [
        (-size, 0),
        (-size / 2, -(3**0.5) / 2 * size),
        (size / 2, -(3**0.5) / 2 * size),
        (size, 0),
        (size / 2, (3**0.5) / 2 * size),
        (-size / 2, (3**0.5) / 2 * size),
    ]
```

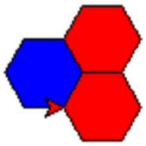
```
    drawer.penup()
    drawer.goto(x, y - 8)
    drawer.write(label, align="center")

    drawer.fillcolor(color)
    drawer.begin_fill()
    drawer.goto(x + vertices[0][0], y + vertices[0][1])
    drawer.pendown()

    for vertex in vertices[1:]:
        drawer.goto(x + vertex[0], y + vertex[1])

    drawer.goto(x + vertices[0][0], y + vertices[0][1])
    drawer.end_fill()


def calculate_adjacent_centers(centers, visited, cords):
    global size
    x, y = cords

    adjacent = [
        (-3 / 2 * size, -(3**0.5) / 2 * size),
        (0, -(3**0.5) * size),
        (3 / 2 * size, -(3**0.5) / 2 * size),
        (3 / 2 * size, (3**0.5) / 2 * size),
        (0, (3**0.5) * size),
        (-3 / 2 * size, (3**0.5) / 2 * size),
    ]

    adjacent_centers = [
        round_position((center[0] + x, center[1] + y)) for center in adjacent
    ]

    for center in adjacent_centers:
        if center not in visited:
            distance = round((center[0] ** 2) + (center[1] ** 2), 4)
            centers.append(center)


def check_input(cords, visited, input_clr):
    x, y = cords

    adjacent = [
        (-3 / 2 * size, -(3**0.5) / 2 * size),
        (0, -(3**0.5) * size),
```

```
        (3 / 2 * size, -(3**0.5) / 2 * size),
        (3 / 2 * size, (3**0.5) / 2 * size),
        (0, (3**0.5) * size),
        (-3 / 2 * size, (3**0.5) / 2 * size),
    ]

    adjacent_centers = [
        round_position((center[0] + x, center[1] + y)) for center in adjacent
    ]

    center_ind = visited.index(cords)

    for center in adjacent_centers:
        ind = visited.index(center)

        if input_clr[center_ind] == input_clr[ind]:
            return False

    return True



def draw_cluster(n):
    global cluster, label

    if n == 0:
        return

    temp = n
    drawer = turtle.Turtle()
    drawer.speed(0)

    centers = []
    centers.append((0, 0))
    visited = []
    cluster = [(label, (0, 0))]

    while temp != 0:
        cords = centers.pop(0)

        if cords not in visited:
            visited.append(cords)
            draw_hexagon(drawer, cords)
            label += 1
```

```
            cluster.append((label, (0, 0)))
            temp -= 1
            calculate_adjacent_centers(centers, visited, cords)

    input_clr = input_colors(n)

    for i in range(n):
        color_hexagon(drawer, visited[i], input_clr[i])

    for i in range(n):
        if not check_input(visited[i], visited, input_clr):
            print("Invalid color selection!")
            break



def input_colors(n):
    input_clr = []

    for i in range(n):
        input_clr.append(input("Enter color for {} cell: ".format(i)))

    return input_clr


def random_color():
    r = random.random()
    g = random.random()
    b = random.random()
    return (r, g, b)


def main():
    i, j = map(int, input("Enter values of i, j: ").split())
    n = i ** 2 + i * j + j ** 2

    screen = turtle.Screen()

    draw_cluster(n)

    screen.exitonclick()


if __name__ == "__main__":
    main()
```

**End Result:**

```
PROBLEMS  OUTPUT  TERMINAL  A

PS E:\GIt> python -u "e:\GIt\SE
Enter values of i, j: 1 1
Enter color for 0 cell: red
Enter color for 1 cell: blue
Enter color for 2 cell: red
Invalid color selection!
```



```
PROBLEMS  OUTPUT  TERMINAL  AZU

PS E:\GIt> python -u "e:\GIt\SEM-
Enter values of i, j: 1 1
Enter color for 0 cell: red
Enter color for 1 cell: blue
Enter color for 2 cell: green
```

**Conclusion:**

Each cellular base station is allocated a group of radio channels to be used within a small geographic

area called a cell. Base stations in adjacent cells are assigned channel groups which contain completely different channels than neighboring cells. Base station antennas are designed to achieve the desired coverage within a particular cell. By limiting the coverage area within the boundaries of a cell, the same group of channels may be used to cover different cells that are separated from one another by geographic distances large enough to keep interference levels within tolerable limits. The design process of selecting and allocating channel groups for all cellular base stations within a system is called frequency reuse or frequency planning