Laporan Tugas Kecil 1
Strategi Algoritma

Timothy Bernard Soeharto
NIM 13524092

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

A. ALGORITMA BRUTE FORCE

Pseudocode:

```
FUNCTION indexToPos(idx, nCol):
   row = idx / nCol        // Integer division
   col = idx % nCol        // Remainder
   RETURN [row, col]

FUNCTION bruteForce(grid, nCol, nRow, colors):
   numQueens = length(colors)
   totalCells = nRow * nCol
   found = false
   result = null

   FUNCTION generate(start, current):
      IF found THEN return

      IF length(current) == numQueens THEN
         IF checkCombination(current, grid, colors) THEN
            result = current
            found = true
         END IF
         return
      END IF

      FOR i = start TO totalCells - 1:
         pos = indexToPos(i)         // Convert i to [row, col]
         current.append(pos)         // Add position to current combination
         generate(i + 1, current)    // Recurse with next starting index
         current.removeLast()        // Remove last position (try next)
      END FOR

   END FUNCTION

   generate(0, [])
   RETURN result

FUNCTION checkCombination(positions, grid, colors):
   FOR every pair (i, j) in positions:
      IF queens are adjacent (touching):
         RETURN false
      IF queens are in same row:
         RETURN false
```

```
    IF queens are in same column:
        RETURN false
END FOR

FOR every queen position:
    IF color already used by another queen:
        RETURN false
    ELSE:
        Mark color as used
END FOR

RETURN true
```

Langkah-langkah algoritma:

1. **Representasi posisi**
   Semua sel pada grid dianggap sebagai satu baris indeks dari 0 sampai totalCells − 1.
   Contoh grid 3×3:
   0 1 2
   3 4 5
   6 7 8

   Konversi index ke baris dan kolom (dengan fungsi indexToPos):
   row = index / nCol
   col = index % nCol

2. **Membuat kombinasi (fungsi generate)**
   Fungsi generate menyusun posisi ratu satu per satu.

   Parameter:
   start = supaya kombinasi tidak terulang, tujuannya untuk merekursif cabang “indeks”
   current = daftar posisi ratu sementara, kalau jumlah posisi ratu sudah sama dengan jumlah warna, akan cek kombinasi (langkah selanjutnya)

   Setiap kali jumlah posisi yang ada di list positions sudah sama dengan jumlah warna, kombinasi tersebut dicek.

3. **Pengecekan kombinasi (fungsi checkCombination)**
   Sebuah kombinasi valid jika:
   - Tidak ada dua ratu bersebelahan (termasuk diagonal)
   - Tidak ada dua ratu di baris yang sama
   - Tidak ada dua ratu di kolom yang sama
   - Setiap ratu berada pada warna yang berbeda

Jika salah satu aturan dilanggar, kombinasi ditolak, lalu menjalankan fungsi generate untuk menghasilkan kombinasi selanjutnya untuk dicek.

4. **Menghentikan proses**
   Begitu ditemukan kombinasi yang valid, variabel “found” diubah menjadi true, lalu semua rekursi langsung berhenti, hasil direturn.

B. Source code (dalam bahasa Golang)

```go
package main

import (
    "fmt"
    "image"
    "image/color"
    _ "image/jpeg"
    "math"
    "os"
    "slices"
    "time"

    "github.com/disintegration/imaging"
)

const queenImagePath = "queen.png"
const cellSize = 100
const gridLineSize = 2

func printGrid(grid [][]byte) {
    for i := 0; i < len(grid); i++ {
        for j := 0; j < len(grid[i]); j++ {
            fmt.Printf("%c ", grid[i][j])
        }
        fmt.Println()
    }
}

func printGridResult(grid [][]byte, positions [][2]int, rowCount,
colCount int) {
    for i := 0; i < rowCount; i++ {
        for j := 0; j < colCount; j++ {
            isQueen := false
            for _, pos := range positions {
                if pos[0] == i && pos[1] == j {
                    isQueen = true
                    break
                }
            }
            if isQueen {
                fmt.Print("# ")
            } else {
                fmt.Printf("%c ", grid[i][j])
            }
        }
        fmt.Println()
    }
}

func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}

func checkCombination(positions [][2]int, grid [][]byte, colors
[]byte) bool {
    copyColors := make([]byte, len(colors))
    copy(copyColors, colors)

    for i := 0; i < len(positions); i++ {
        for j := i + 1; j < len(positions); j++ {
            row1, col1 := positions[i][0], positions[i][1]
            row2, col2 := positions[j][0], positions[j][1]

            // Check if adjacent (also diag)
            if abs(row1-row2) <= 1 && abs(col1-col2) <= 1 {
                return false
            }

            if row1 == row2 { // Check if same row
                return false
            }

            if col1 == col2 { // Check if same col
                return false
            }
        }
    }

    // Check if there is only one queen per color
    for i := 0; i < len(positions); i++ {
        row, col := positions[i][0], positions[i][1]
        color := grid[row][col]

        if slices.Contains(copyColors, color) {
            // Remove the color from copyColors
            for idx, v := range copyColors {
                if v == color {
                    copyColors = append(copyColors[:idx],
copyColors[idx+1:]...)
                    break
                }
            }
        } else { // If the color is not in copyColors anymore, that
means it has been used by another queen
            return false
        }
    }

    return true
}

func bruteForce(grid [][]byte, nCol, nRow int, colors []byte)
([][2]int, int) {
    numQueens := len(colors)
    totalCells := nRow * nCol

    iterCount := 0

    // Helper function to convert index to [row, col], for example
index = 5 with nCol = 2, then generates position [2,1]
    // For generating every possible combination without repeating,
like [[0,1],[1,0]] is identical to [[1,0],[0,1]]
    indexToPos := func(idx int) [2]int {
        return [2]int{idx / nCol, idx % nCol}
    }

    var result [][2]int
    found := false

    // Generate combinatios using recursive function, with the help
of indexToPos
    var generate func(start int, current [][2]int)
    generate = func(start int, current [][2]int) {
        if found {
            return // Stop if solution already found
        }

        if len(current) == numQueens {
            iterCount++

            if iterCount%10000000 == 0 {
                fmt.Printf("\n%d combinations checked\n", iterCount)
                printGridResult(grid, current, nRow, nCol)
            }

            if checkCombination(current, grid, colors) {
                result = make([][2]int, numQueens)
                copy(result, current)
                found = true
            }
            return
```

```go
        }

        for i := start; i < totalCells; i++ {
            pos := indexToPos(i)
            current = append(current, pos)
            generate(i+1, current)
            current = current[:len(current)-1]
        }
    }

    fmt.Printf("Starting brute force search for %d queens on %dx%d
grid...\n", numQueens, nRow, nCol)
    generate(0, [][2]int{})

    return result, iterCount
}

func saveResultAsTxt(grid [][]byte, positions [][2]int, rowCount,
colCount int, iterCount int, elapsed time.Duration, outputPath
string) error {
    file, err := os.Create(outputPath)
    if err != nil {
        return err
    }
    defer file.Close()

    // Write grid with queens
    for i := 0; i < rowCount; i++ {
        for j := 0; j < colCount; j++ {
            isQueen := false
            for _, pos := range positions {
                if pos[0] == i && pos[1] == j {
                    isQueen = true
                    break
                }
            }
            if isQueen {
                fmt.Fprintf(file, "#")
            } else {
                fmt.Fprintf(file, "%c", grid[i][j])
            }
        }
        fmt.Fprintln(file)
    }

    // Write stats
    fmt.Fprintf(file, "Time elapsed: %d ms\n",
elapsed.Milliseconds())
    fmt.Fprintf(file, "Number of combinations tried: %d
combinations\n", iterCount)

    return nil
}

func colorDistance(c1, c2 [3]uint8) float64 {
    dr := float64(c1[0]) - float64(c2[0])
    dg := float64(c1[1]) - float64(c2[1])
    db := float64(c1[2]) - float64(c2[2])
    return math.Sqrt(dr*dr + dg*dg + db*db)
}

func readGridFromImage(filename string, n int) ([][]byte, int,
[]byte, map[byte]color.RGBA, error) {
    file, err := os.Open(filename)
    if err != nil {
        return nil, 0, nil, nil, err
    }
    defer file.Close()

    img, _, err := image.Decode(file)
    if err != nil {
        return nil, 0, nil, nil, err
    }

    bounds := img.Bounds()
    width := bounds.Max.X - bounds.Min.X
    height := bounds.Max.Y - bounds.Min.Y

    cellW := width / n
    cellH := height / n
    sampleCellSize := int(math.Min(float64(cellW), float64(cellH)))

    grid := make([][]byte, n)
    for i := range grid {
        grid[i] = make([]byte, n)
    }

    colorMap := make(map[[3]uint8]byte)
    charToColor := make(map[byte]color.RGBA)
    colors := []byte{}
    nextColorChar := byte('A')

    threshold := 30.0

    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            cx := j*sampleCellSize + sampleCellSize/2
            cy := i*sampleCellSize + sampleCellSize/2

            r, g, b, _ := img.At(cx, cy).RGBA()
            sampledColor := [3]uint8{uint8(r >> 8), uint8(g >> 8),
uint8(b >> 8)}

            // Find closest existing color
            foundChar := byte(0)
            minDist := math.MaxFloat64
            for existingColor, char := range colorMap {
                dr := float64(sampledColor[0]) -
float64(existingColor[0])
                dg := float64(sampledColor[1]) -
float64(existingColor[1])
                db := float64(sampledColor[2]) -
float64(existingColor[2])
                dist := math.Sqrt(dr*dr + dg*dg + db*db)
                if dist < minDist {
                    minDist = dist
                    foundChar = char
                }
            }

            if minDist < threshold {
                grid[i][j] = foundChar
            } else {
                colorMap[sampledColor] = nextColorChar
                charToColor[nextColorChar] =
color.RGBA{sampledColor[0], sampledColor[1], sampledColor[2], 255}
                grid[i][j] = nextColorChar
                colors = append(colors, nextColorChar)
                nextColorChar++
            }
        }
    }

    return grid, n, colors, charToColor, nil
}

func buildCharToColor() map[byte]color.RGBA { // If input is txt, no
color saved to use in save as image.
    return map[byte]color.RGBA{
        'A': {255, 107, 107, 255}, // Red
        'B': {107, 159, 255, 255}, // Blue
        'C': {107, 255, 107, 255}, // Green
        'D': {255, 215, 0, 255},   // Yellow
        'E': {196, 107, 255, 255}, // Purple
        'F': {255, 165, 0, 255},   // Orange
        'G': {255, 105, 180, 255}, // Pink
        'H': {107, 255, 215, 255}, // Teal
        'I': {160, 82, 45, 255},   // Brown
        'J': {0, 255, 255, 255},   // Cyan
        'K': {255, 20, 147, 255},  // Deep Pink
        'L': {50, 205, 50, 255},   // Lime Green
        'M': {255, 140, 0, 255},   // Dark Orange
```

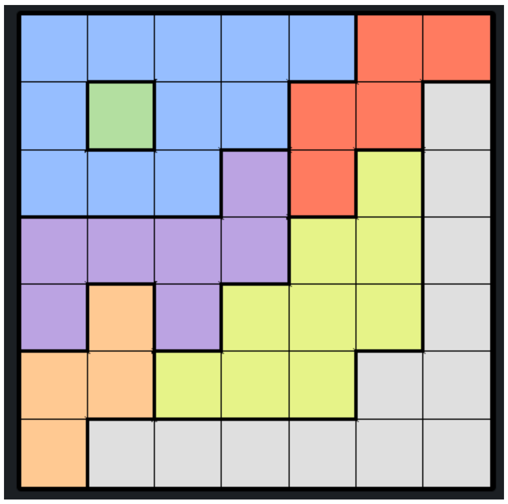
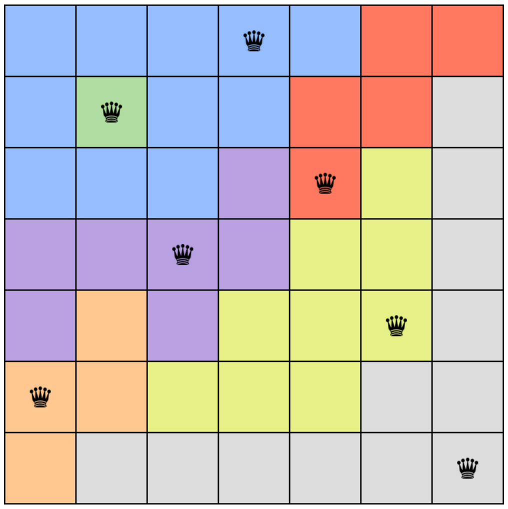
```go
        'N': {0, 191, 255, 255},   // Deep Sky Blue
        'O': {148, 0, 211, 255},   // Dark Violet
        'P': {0, 128, 128, 255},   // Dark Teal
        'Q': {220, 20, 60, 255},   // Crimson
        'R': {127, 255, 0, 255},   // Chartreuse
        'S': {255, 228, 196, 255}, // Bisque
        'T': {70, 130, 180, 255},  // Steel Blue
        'U': {244, 164, 96, 255},  // Sandy Brown
        'V': {0, 255, 127, 255},   // Spring Green
        'W': {255, 99, 71, 255},   // Tomato
        'X': {123, 104, 238, 255}, // Medium Slate Blue
        'Y': {255, 255, 102, 255}, // Light Yellow
        'Z': {64, 224, 208, 255},  // Turquoise
    }
}

func saveResultAsImage(grid [][]byte, positions [][2]int, n int,
charToColor map[byte]color.RGBA, outputPath string) error {
    queenFile, err := os.Open(queenImagePath)
    if err != nil {
        return fmt.Errorf("could not open queen image: %v", err)
    }
    defer queenFile.Close() // Close file when functions returns

    queenImg, _, err := image.Decode(queenFile) // Like algeo, decode
the image to become a matriks of colors
    if err != nil {
        return fmt.Errorf("could not decode queen image: %v", err)
    }

    queenSize := int(float64(cellSize) * 0.30) // Queen size is 30%
of cell size
    queenResized := imaging.Resize(queenImg, queenSize, queenSize,
imaging.Lanczos)

    totalSize := n*cellSize + (n+1)*gridLineSize
    outputImg := imaging.New(totalSize, totalSize, color.RGBA{0, 0,
0, 255}) // Black background (grid lines)
```

```
    // Draw each cell
    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            cellColor := charToColor[grid[i][j]]
            startX := j*cellSize + (j+1)*gridLineSize
            startY := i*cellSize + (i+1)*gridLineSize
            cellImg := imaging.New(cellSize, cellSize, cellColor)
            outputImg = imaging.Paste(outputImg, cellImg,
image.Pt(startX, startY))
        }
    }

    // Overlay queen on each queen position
    for _, pos := range positions {
        row, col := pos[0], pos[1]
        cellX := col*cellSize + (col+1)*gridLineSize
        cellY := row*cellSize + (row+1)*gridLineSize
        queenX := cellX + (cellSize-queenSize)/2
        queenY := cellY + (cellSize-queenSize)/2
        outputImg = imaging.Overlay(outputImg, queenResized,
image.Pt(queenX, queenY), 1.0)
    }

    return imaging.Save(outputImg, outputPath)
}

func main() {
    var inputType string
    fmt.Print("Input type (text/image): ")
    fmt.Scan(&inputType)

    var grid [][]byte
    var rowCount, colCount int
    var colors []byte
    var charToColor map[byte]color.RGBA

    if inputType == "image" {
        var fileName string
        fmt.Print("Enter image filename (in test/input/): ")
        fmt.Scan(&fileName)

        var n int
        fmt.Print("Enter board size (n for nxn board): ")
        fmt.Scan(&n)

        var err error
        grid, rowCount, colors, charToColor, err =
readGridFromImage("../test/input/"+fileName, n)
        if err != nil {
            panic(err)
        }

        err = saveDebugImage("../test/input/"+fileName, n,
"../test/output/debug.png")
        if err != nil {
            fmt.Printf("Warning: could not save debug image: %v\n",
err)
        } else {
            fmt.Println("Debug image saved to
../test/output/debug.png")
        }

        colCount = rowCount

        fmt.Println("Grid read from image:")
        printGrid(grid)

    } else {
        var fileName string
        fmt.Print("Enter filename (make sure file is in test/input):
")
        fmt.Scan(&fileName)
        dataBytes, err := os.ReadFile("../test/input/" + fileName)

        if err != nil {
```

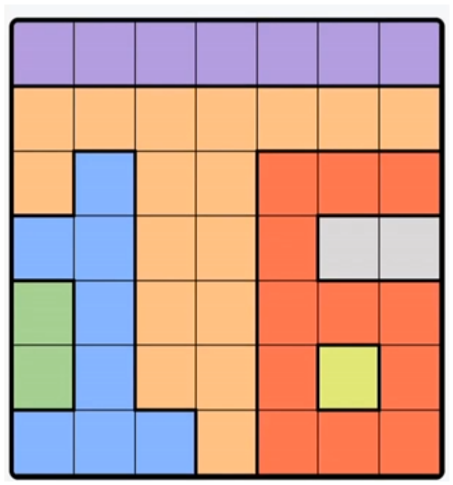
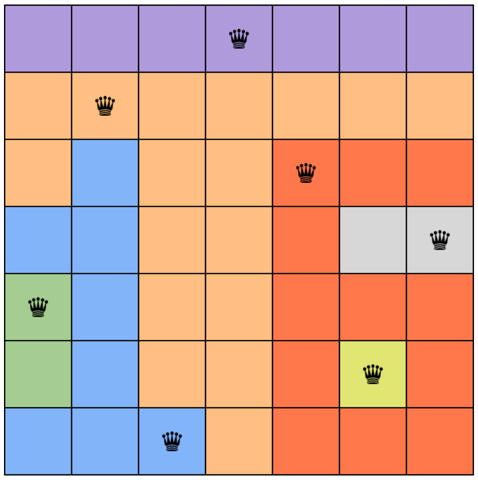
```go
            panic(err)
        }

        inputString := string(dataBytes)
        fmt.Println(inputString)

        if inputString != "" {
            colors = []byte{}
            colCounter := 0
            colCount = -1
            rowCount = 0
            for i := 0; i < len(inputString); i++ {
                if inputString[i] == '\r' {
                    continue
                }
                if inputString[i] == '\n' {
                    rowCount++
                    if colCount == -1 {
                        colCount = colCounter
                    } else {
                        if colCount != colCounter {
                            fmt.Println("Error: Inconsistent column
dimension")
                            return
                        }
                    }
                    colCounter = 0
                    continue
                }
                if !slices.Contains(colors, inputString[i]) {
                    colors = append(colors, inputString[i])
                }
                colCounter++
            }
            if colCounter > 0 {
                if colCount == -1 {
                    colCount = colCounter
                } else if colCount != colCounter {
                    fmt.Println("Error: Inconsistent column
dimension")
                    return
                }
            }
            if len(inputString) > 0 &&
inputString[len(inputString)-1] != '\n' {
                rowCount++
            }

            grid = make([][]byte, rowCount)
            for i := range grid {
                grid[i] = make([]byte, colCount)
            }

            index := 0
            for i := 0; i < rowCount; i++ {
                for j := 0; j < colCount; j++ {
                    for inputString[index] == '\n' ||
inputString[index] == '\r' {
                        index++
                    }
                    grid[i][j] = inputString[index]
                    index++
                }
            }

            printGrid(grid)
            charToColor = buildCharToColor() // Generate the map from
character to color if input is text instead of image
        } else {
            fmt.Println("No text file content")
            return
        }
    }

    for i := 0; i < len(colors); i++ {
        fmt.Printf("%c\n", colors[i])
    }
    fmt.Printf("Amount of color: %d\n", len(colors))
    fmt.Printf("Dimension : %dx%d\n", rowCount, colCount)

    if len(colors) > rowCount {
        fmt.Println("Error: More colors than rowCount/colCount, not
possible")
        return
    }

    startTime := time.Now()
    result, iterCount := bruteForce(grid, colCount, rowCount, colors)
    elapsed := time.Since(startTime)

    if result != nil {
        fmt.Printf("\n\nSearch time: %d ms\n",
elapsed.Milliseconds())
        fmt.Printf("Number of combinations tried: %d combinations\n",
iterCount)
        fmt.Printf("Result:\n")
        printGridResult(grid, result, rowCount, colCount)

        // Ask for txt
        var saveTxt string
        fmt.Print("\nDo you want to save results to txt? (y/n): ")
        fmt.Scan(&saveTxt)
        if saveTxt == "y" {
            fmt.Print("\nInput filename for .txt file : ")
            var txtFileName string
            fmt.Scan(&txtFileName)
            err := saveResultAsTxt(grid, result, rowCount, colCount,
iterCount, elapsed, "../test/output/"+txtFileName+".txt")
            if err != nil {
                fmt.Printf("Error saving txt: %v\n", err)
            } else {
                fmt.Println(".txt file saved to filepath
../test/output/" + txtFileName + ".txt")
            }
        }

        // Ask for image
        var saveImg string
        fmt.Print("Do you want to save results to image? (y/n): ")
        fmt.Scan(&saveImg)
        if saveImg == "y" {
            fmt.Print("\nInput filename for image file : ")
            var imgFileName string
            fmt.Scan(&imgFileName)
            err := saveResultAsImage(grid, result, rowCount,
charToColor, "../test/output/"+imgFileName+".png")
            if err != nil {
                fmt.Printf("Error saving image: %v\n", err)
            } else {
                fmt.Println("Image saved to filepath ../test/output/"
+ imgFileName + ".png")
            }
        }
    } else {
        fmt.Printf("\n\nNo solution found after %d iterations.\n",
iterCount)
        fmt.Printf("Time elapsed: %d ms\n", elapsed.Milliseconds())
    }
}
```

C. Input dan Output (Testcases)

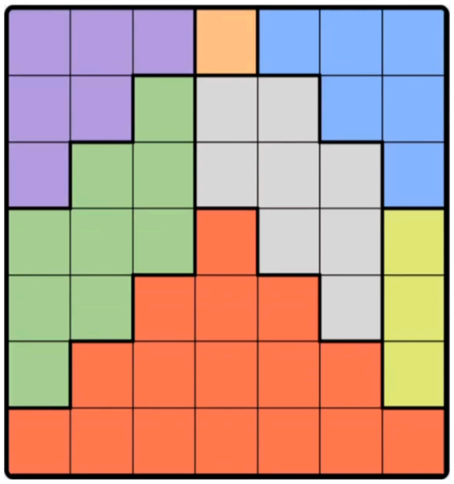
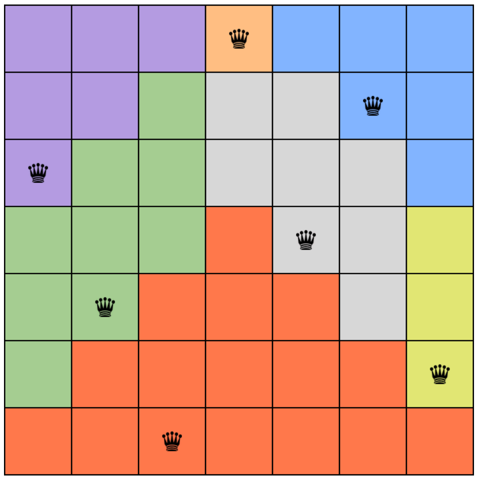
Input & Output 1:

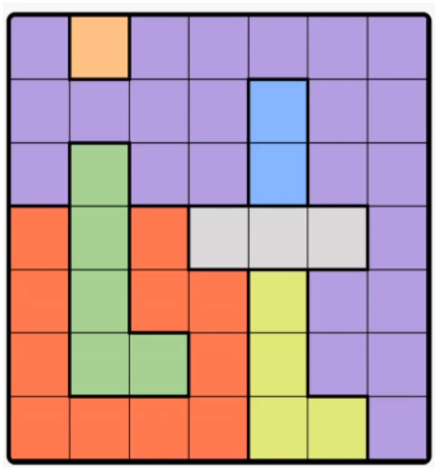
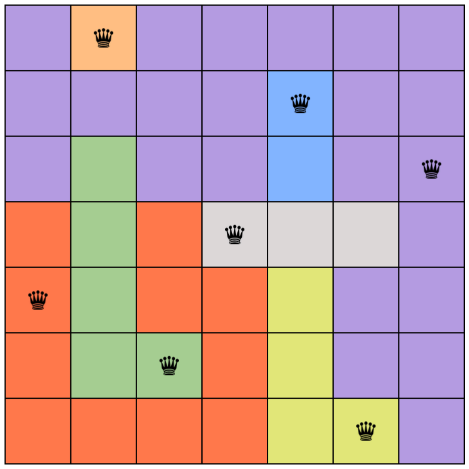
Input & Output 2:

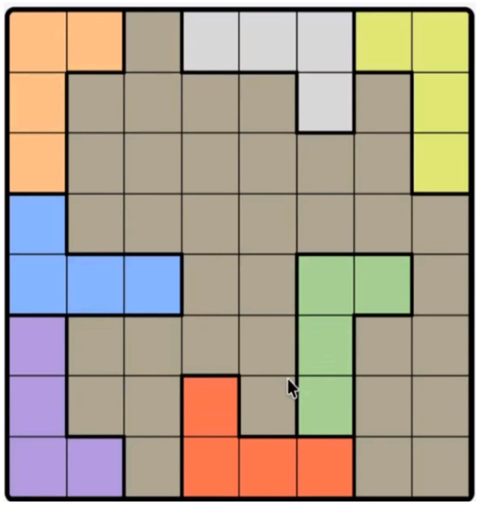
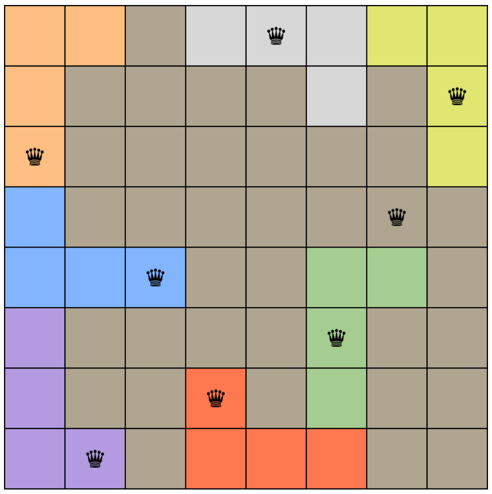
Input & Output 3:

Input & Output 4:

Input & Output 5:

D. Link Repository Github

https://github.com/tmthyberd/Tucil1_13524092

E. Pernyataan

Tugas ini disusun sepenuhnya tanpa bantuan kecerdasan buatan (Generative AI), melainkan hasil pemikiran dan analisis mandiri.

[illegible]

Timothy Bernard Soeharto

LAMPIRAN

No.	Poin	Ya	Tidak
1.	Program berhasil di kompilasi tanpa kesalahan	✓	
2.	Program berhasil di jalankan	✓	
3.	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4.	Program dapat membaca masukan berkas .txt serta menyimpan solusi dalam berkas .txt	✓	
5.	Program memiliki Graphical User Interface (GUI)		✓
6.	Program dapat menyimpan solusi dalam bentuk file gambar	✓	