Федеральное государственное автономное образовательное
учреждение высшего образования
Университет ИТМО

Факультет инфокоммуникационных технологий

Алагритмы и структуры данных:

Отчёт по лабораторной работе №3: Графы

Выполнил:
Бочкарь Артём Артёмович

Группа: **K33392**
Вариант: **13**

Преподаватели:
Артамонова В. Е.

Санкт-Петербург 2023 г.

Введение:
Программы реализовывал на языке Swift. Ввод и вывод реализовал через консоль по причине того, что использовал web-версию среды разработки Swift: среда.
Тесты реализовал с помощью следующей функции:

```
import Foundation

// Функция для измерения времени выполнения кода
func measureTime(block: () -> Void) -> TimeInterval {
    let startTime = Date()
    block()
    let endTime = Date()
    return endTime.timeIntervalSince(startTime)
}

// Пример использования
let timeTaken = measureTime {
    // Код, время выполнения которого нужно измерить
}

print("Время выполнения: \(timeTaken) секунд")
```

Задача №1: Лабиринт (1 балл)

Лабиринт представляет собой прямоугольную сетку ячеек со стенками между некоторыми соседними ячейками.
Вы хотите проверить, существует ли путь от данной ячейки к данному выходу из лабиринта, где выходом также является ячейка, лежащая на границе лабиринта (в примере, показанном на рисунке, есть два выхода: один на левой границе и один на правой границе). Для этого вы представляете лабиринт в виде неориентированного графа: вершины графа являются ячейками лабиринта, две вершины соединены неориентированным ребром, если они смежные и между ними нет стены. Тогда, чтобы проверить, существует ли путь между двумя заданными ячейками лабиринта, достаточно проверить, что существует путь между соответствующими двумя вершинами в графе.

Вам дан неориентированный граф и две различные вершины u и v. Проверьте, есть ли путь между u и v.

Формат ввода / входного файла (input.txt).
Неориентированный граф с n вершинами и m ребрами по формату 1. Следующая строка после ввода всго графа содержит две вершины u и v.

Формат вывода / выходного файла (output.txt).
Выведите 1, если есть путь между вершинами u и v; выведите 0, если пути нет.

Код программы:

```
import Foundation

func DFS(graph: [[Int]], visited: inout [Bool], start: Int, end: Int) -> Bool {
```

```
    visited[start] = true

    if start == end {
        return true
    }

    for i in 0..<graph.count {
        if graph[start][i] == 1 && !visited[i] {
            if DFS(graph: graph, visited: &visited, start: i, end: end) {
                return true
            }
        }
    }
    return false
}

func hasPath(graph: [[Int]], start: Int, end: Int) -> Bool {
    var visited = Array(repeating: false, count: graph.count)
    return DFS(graph: graph, visited: &visited, start: start - 1, end: end - 1)
}

let firstLine = readLine()!.split(separator: " ").map { Int($0)! }
let n = firstLine[0]
let m = firstLine[1]

var graph = Array(repeating: Array(repeating: 0, count: n), count: n)

for _ in 0..<m {
    let edge = readLine()!.split(separator: " ").map { Int($0)! }
    let u = edge[0] - 1
    let v = edge[1] - 1
    graph[u][v] = 1
    graph[v][u] = 1
}

let vertices = readLine()!.split(separator: " ").map { Int($0)! }
let u = vertices[0]
let v = vertices[1]

print("")
let startTime = Date()
if hasPath(graph: graph, start: u, end: v) {
    print("1")
} else {
    print("0")
}
let endTime = Date()

let timeElapsedInSeconds = endTime.timeIntervalSince(startTime)
print("Время работы алгоритма: \(timeElapsedInSeconds) секунд")
```

Результат работы программы 1:

```
swift /tmp/AsNyJBte7z.swift
4 4
1 2
3 2
4 3
1 4
1 4

1
Время работы алгоритма: 2.300739288330078e-05 секунд
```

Результат работы программы 2:

```
swift /tmp/3UimUXdOW0.swift
4 2
1 2
3 2
1 4

0
Время работы алгоритма: 2.7060508728027344e-05 секунд
```

Задача №2: Компоненты (1 балл)

Теперь вы решаете сделать так, чтобы в лабиринте не было мертвых зон, то есть чтобы из каждой клетки был доступен хотя бы один выход. Для этого вы находите связные компоненты соответствующего неориентированного графа и следите за тем, чтобы каждый компонент содержал выходную ячейку.
Дан неориентированный граф с n вершинами и m ребрами. Нужно посчитать количество компонент связяности в нем.

Формат ввода / входного файла (input.txt).
Неориентированный граф с n вершинами и m ребрами по формату 1.

Формат вывода / выходного файла (output.txt).
Выведите количество компонент связности.

Код программы:

```
import Foundation

class Graph {
    var vertices: Int
```

```swift
    var edges: Int
    var adjList: [[Int]]
    var visited: [Bool]

    init(vertices: Int, edges: Int) {
        self.vertices = vertices
        self.edges = edges
        self.adjList = Array(repeating: [], count: vertices + 1)
        self.visited = Array(repeating: false, count: vertices + 1)
    }

    func addEdge(_ u: Int, _ v: Int) {
        adjList[u].append(v)
        adjList[v].append(u)
    }

    func dfs(_ v: Int) {
        visited[v] = true
        for u in adjList[v] {
            if !visited[u] {
                dfs(u)
            }
        }
    }

    func countConnectedComponents() -> Int {
        var count = 0
        for v in 1...vertices {
            if !visited[v] {
                count += 1
                dfs(v)
            }
        }
        return count
    }
}

func readInput() -> (Int, Int) {
    guard let input = readLine() else { fatalError("Failed to read input") }
    let values = input.split(separator: " ").map { Int($0)! }
    return (values[0], values[1])
}

let (n, m) = readInput()
let graph = Graph(vertices: n, edges: m)

for _ in 0..<m {
    let edge = readInput()
    graph.addEdge(edge.0, edge.1)
}

let startTime = Date().timeIntervalSince1970
let connectedComponents = graph.countConnectedComponents()
let endTime = Date().timeIntervalSince1970
let elapsedTime = endTime - startTime

print("")
print("Количество компонент связности: \(connectedComponents)")
```

```
print("Время выполнения программы: \(elapsedTime) секунд")
```

Результат работы программы:

```
swift /tmp/K6IWsbcvsh.swift
4 2
1 2
3 2

Количество компонент связности: 2
Время выполнения программы: 0.00011897087097167969 секунд
```

Задача №3: Циклы (1 балл)

Учебная программа по инфокоммуникационным технологиям определяет пререквизиты для каждого курса в виде списка курсов, которые необходимо пройти перед тем, как начать этот курс. Вы хотите выполнить проверку согласованности учебного плана, то есть проверить отсутствие циклических зависимостей. Для этого строится следующий ориентированный граф: вершины соответствуют курсам, есть направленное ребро (u, v) – курс u следует пройти перед курсом v. Затем достаточно проверить, содержит ли полученный граф цикл.
Проверьте, содержит ли данный граф циклы.

Формат ввода / входного файла (input.txt).
Ориентированный граф с n вершинами и m ребрами по формату 1.

Формат вывода / выходного файла (output.txt).
Выведите 1, если данный граф содержит цикл; выведите 0, если не содержит.

Код программы:

```
import Foundation

// Структура для представления графа
struct Graph {
    let vertices: Int
    var adjList: [[Int]]

    init(vertices: Int) {
        self.vertices = vertices
        adjList = Array(repeating: [], count: vertices)
    }

    mutating func addEdge(_ u: Int, _ v: Int) {
        adjList[u].append(v)
    }

    // Функция для проверки цикла
```

```swift
    func hasCycle() -> Bool {
        var visited = Array(repeating: false, count: vertices)
        var recursionStack = Array(repeating: false, count: vertices)

        func isCyclic(_ v: Int) -> Bool {
            if !visited[v] {
                visited[v] = true
                recursionStack[v] = true

                for neighbour in adjList[v] {
                    if !visited[neighbour] && isCyclic(neighbour) {
                        return true
                    } else if recursionStack[neighbour] {
                        return true
                    }
                }
            }

            recursionStack[v] = false
            return false
        }

        for vertex in 0..<vertices {
            if isCyclic(vertex) {
                return true
            }
        }

        return false
    }
}

// Ввод графа
func inputGraph() -> Graph? {
    if let input = readLine() {
        let params = input.split(separator: " ").compactMap { Int($0) }
        let vertices = params[0]
        let edges = params[1]
        var graph = Graph(vertices: vertices)

        for _ in 0..<edges {
            if let edgeInput = readLine() {
                let edgeParams = edgeInput.split(separator: " ").compactMap { Int($0) }
                graph.addEdge(edgeParams[0] - 1, edgeParams[1] - 1)
            }
        }

        return graph
    }

    return nil
}

if let graph = inputGraph() {
    let startTime = DispatchTime.now()
    let hasCycle = graph.hasCycle()
    let endTime = DispatchTime.now()
    let timeInterval = Double(endTime.uptimeNanoseconds - startTime.uptimeNanoseconds) / 1_000_000_000
```

```
    print("")

    if hasCycle {
        print("1")
    } else {
        print("0")
    }
    print("Время работы алгоритма: \(timeInterval) секунд")
}
```

Результат работы программы 1:

```
swift /tmp/Kmi1JNZonk.swift
4 4
1 2
4 1
2 3
3 1

1
Время работы алгоритма: 0.00014663 секунд
```

Результат работы программы 2:

```
swift /tmp/NYb2zBcBY4.swift
5 7
1 2
2 3
1 3
3 4
1 4
2 5
3 5

0
Время работы алгоритма: 0.00012984 секунд
```

Задача №4: Порядок курсов (1 балл)

Теперь, когда вы уверены, что в данном учебном плане нет циклических зависимостей, вам нужно найти порядок всех курсов, соответствующий всем зависимостям. Для

этого нужно сделать топологическую сортировку соответству-
ющего ориентированного графа.
Дан ориентированный ациклический граф (DAG) с n вершинами и m ребрами.
Выполните топологическую сортировку.

Формат ввода / входного файла (input.txt).
Ориентированный ациклический граф с n вершинами и m ребрами по формату 1.

Формат вывода / выходного файла (output.txt).
Выведите любое линейное упорядочение данного графа (Многие ациклические графы имеют более одного варианта упорядочения, вы можете вывести любой из них).

Код программы:

```
import Foundation

func topologicalSort(_ graph: [[Int]]) -> [Int] {
    var inDegree = [Int](repeating: 0, count: graph.count)
    for neighbors in graph {
        for neighbor in neighbors {
            inDegree[neighbor] += 1
        }
    }

    var queue = [Int]()
    for i in 0..<graph.count {
        if inDegree[i] == 0 {
            queue.append(i)
        }
    }

    var result = [Int]()
    while !queue.isEmpty {
        let node = queue.removeFirst()
        result.append(node)
        for neighbor in graph[node] {
            inDegree[neighbor] -= 1
            if inDegree[neighbor] == 0 {
                queue.append(neighbor)
            }
        }
    }

    return result
}

func calculateExecutionTime(block: () -> Void) {
    let startTime = Date()
    block()
    let endTime = Date()
    let executionTime = endTime.timeIntervalSince(startTime)
    print("Время выполнения алгоритма: \(executionTime) секунд")
}

func main() {
    guard let input = readLine() else { return }
```

```
    let values = input.split(separator: " ").compactMap { Int($0) }
    let n = values[0]
    let m = values[1]

    var graph = [[Int]](repeating: [], count: n)

    for _ in 0..<m {
        guard let edgeInput = readLine() else { return }
        let edge = edgeInput.split(separator: " ").compactMap { Int($0) }
        if edge.count == 2 {
            graph[edge[0] - 1].append(edge[1] - 1)
        }
    }

    calculateExecutionTime {
        let result = topologicalSort(graph)
        print("")
        print(result.map { String($0 + 1) }.joined(separator: " "))
    }
}

main()
```

Результат работы программы 1:

```
swift /tmp/q8JbMINO9T.swift
4 3
1 2
4 1
3 1

3 4 1 2
Время выполнения алгоритма: 0.00024199485778808594 секунд
```

Результат работы программы 2:

```
swift /tmp/28BvpBuvCr.swift
4 1
3 1

2 3 4 1
Время выполнения алгоритма: 0.0002700090404083251953 секунд
```

Задача №5: Город с одностороггим движением (1.5 балла)

Департамент полиции города сделал все улицы односторонними. Вы хотели бы проверить, можно ли законно проехать с любого перекрестка на какой-либо другой

перекресток. Для этого строится ориентированный граф: вершины – это перекрестки, существует ребро (u, v) всякий раз, когда в городе есть улица (с односторонним движением) из u в v. Тогда достаточно проверить, все ли вершины графа лежат в одном компоненте сильной связности.
Нужно вычислить количество компонентов сильной связности заданного ориентированного графа с n вершинами и m ребрами.

Формат ввода / входного файла (input.txt).
Ориентированный граф с n вершинами и m ребрами по формату 1.

Формат вывода / выходного файла (output.txt).
Выведите число – количество компонентов сильной связности.

Код программы:

```
import Foundation

// Класс для создания графа и поиска компонентов сильной связности
class StronglyConnectedComponents {
    var graph: [[Int]]
    var n: Int
    var visited: [Bool]
    var reverseGraph: [[Int]]
    var finishTimes: [Int]

    init(n: Int, edges: [[Int]]) {
        self.n = n
        self.graph = Array(repeating: [], count: n)
        self.reverseGraph = Array(repeating: [], count: n)
        self.visited = Array(repeating: false, count: n)
        self.finishTimes = []

        for edge in edges {
            let u = edge[0] - 1
            let v = edge[1] - 1
            graph[u].append(v)
            reverseGraph[v].append(u)
        }
    }

    func explore(node: Int, graph: [[Int]], visited: inout [Bool], finishOrder: inout [Int]) {
        visited[node] = true
        for neigh in graph[node] {
            if !visited[neigh] {
                explore(node: neigh, graph: graph, visited: &visited, finishOrder: &finishOrder)
            }
        }
        finishOrder.append(node)
    }

    func finishTimesDFS() {
        for i in 0..<n {
```

```
            if !visited[i] {
                explore(node: i, graph: graph, visited: &visited, finishOrder: &finishTimes)
            }
        }
    }

    func countSCCs() -> Int {
        var sccCount = 0
        finishTimesDFS()
        visited = Array(repeating: false, count: n)

        for i in finishTimes.reversed() {
            if !visited[i] {
                explore(node: i, graph: reverseGraph, visited: &visited, finishOrder: &finishTimes)
                sccCount += 1
            }
        }
        return sccCount
    }
}

if let input = readLine() {
    let data = input.split(separator: " ").map{Int($0)!}
    let n = data[0]
    let m = data[1]
    var edges: [[Int]] = []

    for _ in 0..<m {
        if let edge = readLine() {
            edges.append(edge.split(separator: " ").map{Int($0)!})
        }
    }

    let scc = StronglyConnectedComponents(n: n, edges: edges)
    let startTime = Date()
    let sccCount = scc.countSCCs()
    let endTime = Date()
    let executionTime = endTime.timeIntervalSince(startTime)
    print("")
    print(sccCount)
    print("Время выполнения алгоритма: \(executionTime) секунд")
}
```

Результат работы программы 1:

```
swift /tmp/TaSf2wHphb.swift
4 4
1 2
4 1
2 3
3 1

2
Время выполнения алгоритма: 0.00010502338409423828 секунд
```

Результат работы программы 2:

```
swift /tmp/hkSvUN4jsg.swift
5 7
2 1
3 2
3 1
4 3
4 1
5 2
5 3

5
Время выполнения алгоритма: 0.00011205673217773438 секунд
```

Задача №6: Количество пересадок (1 балл)

Вы хотите вычислить минимальное количество сегментов полета, чтобы добраться из одного города в другой. Для этого вы строите следующий неориентированный граф: вершины представляют города, между двумя вершинами есть ребро всякий раз, когда между соответствующими двумя городами есть перелет. Тогда достаточно найти кратчайший путь из одного из заданных городов в другой.
Дан неориентироанный граф с n вершинами и m ребрами, а также две вершины u и v, нужно посчитать длину кратчайшего пути между u и v (то есть, минимальное количество ребер в пути из u в v).

Формат ввода / входного файла (input.txt).
Нериентированный граф задан по формату 1. Следующая строка содержит две вершины u и v.

Формат ввода / входного файла (input.txt).
Нериентированный граф задан по формату 1. Следующая строка содержит две вершины u и v.

Код программы:

```
import Foundation

func shortestPath(vertices: Int, edges: Int, u: Int, v: Int, graph: [[Int]]) -> Int {
    var queue: [Int] = []
    var visited: [Bool] = Array(repeating: false, count: vertices + 1)
    var distance: [Int] = Array(repeating: 0, count: vertices + 1)

    queue.append(u)
    visited[u] = true

    while queue.count > 0 {
        let node = queue.removeFirst()

        for neighbor in graph[node] {
            if !visited[neighbor] {
                queue.append(neighbor)
                visited[neighbor] = true
                distance[neighbor] = distance[node] + 1
            }
        }
    }

    return visited[v] ? distance[v] : -1
}

let input = readLine()!.split(separator: " ").map { Int($0)! }
let n = input[0]
let m = input[1]

var graph = Array(repeating: [Int](), count: n + 1)

for _ in 0..<m {
    let edge = readLine()!.split(separator: " ").map { Int($0)! }
    let u = edge[0]
    let v = edge[1]

    graph[u].append(v)
    graph[v].append(u)
}

let uv = readLine()!.split(separator: " ").map { Int($0)! }
let u = uv[0]
let v = uv[1]

let startTime = Date()

let result = shortestPath(vertices: n, edges: m, u: u, v: v, graph: graph)

let endTime = Date()

print("")
```

```
print(result)
print("Время работы алгоритма: \(endTime.timeIntervalSince(startTime))  секунд")
```

Результат работы программы 1:

```
swift /tmp/rIH8IvbsvN.swift
4 4
1 2
4 1
2 3
3 1
2 4

2
Время работы алгоритма: 0.0002009868621826172  секунд
```

Результат работы программы 2:

```
swift /tmp/UkLPqvazNl.swift
5 4
5 2
1 3
3 4
1 4
3 5

-1
Время работы алгоритма: 0.00018894672393798828  секунд
```

Задача №7: Двудольный граф (1.5 балла)

Неориентированный граф называется двудольным, если его вершины можно разбить на две части так, что каждое ребро графа соединяет вершины из разных частей, то есть не существует рёбер между вершинами одной и той же части графа. Двудольные графы естественным образом возникают в задачах, где граф используется для моделирования связей между объектами двух разных типов (например, мальчиками и девочками, или студентами и общежитиями).
Альтернативное определение таково: граф двудольный, если его вершины можно раскрасить двумя цветами (например, черным и белым) так, что концы каждого ребра окрашены в разные цвета.
Дан неориентированный граф с n вершинами и m ребрами, проверьте, является ли он двудольным.

Формат ввода / входного файла (input.txt).
Неориентированный граф задан по формату 1.

Формат вывода / выходного файла (output.txt).
Выведите 1, если граф двудольный; и 0 в противном случае.

Код программы:

```swift
import Foundation

// Структура для представления ребра графа
struct Edge {
    var start: Int
    var end: Int
}

// Функция для проверки двудольности графа с помощью BFS
func isBipartiteGraph(_ n: Int, _ edges: [Edge]) -> Bool {
    var graph = [[Int]](repeating: [], count: n + 1)

    // Строим граф
    for edge in edges {
        graph[edge.start].append(edge.end)
        graph[edge.end].append(edge.start)
    }

    var colors = [Int](repeating: 0, count: n + 1)
    let visited = [Bool](repeating: false, count: n + 1)

    // Функция для BFS
    func bfs(_ node: Int) -> Bool {
        var queue = [Int]()
        queue.append(node)
        colors[node] = 1

        while !queue.isEmpty {
            let current = queue.removeFirst()

            for neighbor in graph[current] {
                if colors[neighbor] == 0 {
                    colors[neighbor] = colors[current] == 1 ? 2 : 1
                    queue.append(neighbor)
                } else if colors[neighbor] == colors[current] {
                    return false
                }
            }
        }

        return true
    }

    for i in 1...n {
        if !visited[i] {
            if !bfs(i) {
                return false
            }
        }
    }

    return true
```

```swift
}

// Ввод данных
print("Введите количество вершин n и количество ребер m:")
if let input = readLine() {
    let values = input.split(separator: " ").map { Int($0)! }
    let n = values[0]
    let m = values[1]

    print("Введите ребра графа:")
    var edges = [Edge]()
    for _ in 0..<m {
        if let edgeInput = readLine() {
            let edgeValues = edgeInput.split(separator: " ").map { Int($0)! }
            edges.append(Edge(start: edgeValues[0], end: edgeValues[1]))
        }
    }

    // Проверка является ли граф двудольным
    let startTime = Date()
    let isBipartite = isBipartiteGraph(n, edges)
    let endTime = Date()
    let executionTime = endTime.timeIntervalSince(startTime)

    // Вывод результата
    print("")
    print(isBipartite ? "1" : "0")
    print("Время выполнения алгоритма: \(executionTime) секунд")
}
```

Результат работы программы 1:

```
swift /tmp/IEqAnnEbr7.swift
Введите количество вершин n и количество ребер m:
4 4
Введите ребра графа:
1 2
4 1
2 3
3 1

0
Время выполнения алгоритма: 0.00060594081878662 11 секунд
```

Результат работы программы 2:

```
swift /tmp/ydSmh3RCST.swift
Введите количество вершин n и количество ребер m:
5 4
Введите ребра графа:
5 2
4 2
3 4
1 4

1
Время выполнения алгоритма: 0.00039696693420410156 секунд
```

Задача №12: Цветной лабиринт (2 балл)

В одном из парков одного большого города недавно был организован новый аттракцион Цветной лабиринт. Он состоит из n комнат, соединенных m двунаправленными коридорами. Каждый из коридоров покрашен в один из ста цветов, при этом от каждой комнаты отходит не более одного коридора каждого цвета. При этом две комнаты могут быть соединены любым количеством коридоров.
Человек, купивший билет на аттракцион, оказывается в комнате номер один. Кроме билета, он также получает описание пути, по которому он может выбраться из лабиринта. Это описание представляет собой последовательность цветов c1 . . . ck. Пользоваться ей надо так: находясь в комнате, надо посмотреть на очередной цвет в этой последовательности, выбрать коридор такого цвета и пойти по нему. При этом если из комнаты нельзя пойти по коридору соответствующего цвета, то человеку приходится дальше самому выбирать, куда идти.
В последнее время в администрацию парка стали часто поступать жалобы от заблудившихся в лабиринте людей. В связи с этим, возникла необходимость написания программы, проверяющей корректность описания и пути, и, в случае ее корректности, сообщающей номер комнаты, в которую ведет путь.
Описание пути некорректно, если на пути, который оно описывает, возникает ситуация, когда из комнаты нельзя пойти по коридору соответствующего цвета.

Формат входных данных (input.txt) и ограничения.
Первая строка входного файла INPUT.TXT содержит два целых числа n ($1 \le n \le 10000$) и m ($1 \le m \le 100000$) - соответственно количество комнат и коридоров в лабиринте. Следующие m строк содержат описания коридоров. Каждое описание содержит три числа u ($1 \le u \le n$), v ($1 \le v \le n$), c ($1 \le c \le 100$) - соответственно номера комнат, соединенных этим коридором, и цвет коридора. Следующая, (m + 2)-ая строка входного файла содержит длину описания пути - целое число k ($0 \le k \le 100000$). Последняя строка входного файла содержит k целых чисел, разделенных пробелами, - описание пути по лабиринту.

Формат выходных данных (output.txt).
В выходной файл OUTPUT.TXT выведите строку INCORRECT, если описание пути некорректно, иначе выведите номер комнаты, в которую ведет описанный путь. Помните, что путь начинается в комнате номер один.

Код программы:

```
import Foundation

// Structure to represent a corridor
struct Corridor {
    let room1: Int
    let room2: Int
    let color: Int
}

// Main function to check the validity of the path
func checkPathValidity(n: Int, m: Int, corridors: [Corridor], pathLength: Int, path: [Int]) {
    var currentRoom = 1

    for color in path {
        var foundCorridor = false

        for corridor in corridors {
            if corridor.room1 == currentRoom && corridor.color == color {
                currentRoom = corridor.room2
                foundCorridor = true
                break
            } else if corridor.room2 == currentRoom && corridor.color == color {
                currentRoom = corridor.room1
                foundCorridor = true
                break
            }
        }

        if !foundCorridor {
            print("")
            print("INCORRECT")
            return
        }
    }

    print("")
    print(currentRoom)
}

// Read input
if let input = readLine() {
    let parts = input.components(separatedBy: " ")
    let n = Int(parts[0]) ?? 0
    let m = Int(parts[1]) ?? 0

    var corridors = [Corridor]()
    for _ in 0..<m {
        if let corridorInput = readLine() {
            let corridorParts = corridorInput.components(separatedBy: " ")
```

```
            let room1 = Int(corridorParts[0]) ?? 0
            let room2 = Int(corridorParts[1]) ?? 0
            let color = Int(corridorParts[2]) ?? 0
            corridors.append(Corridor(room1: room1, room2: room2, color: color))
        }
    }

    if let pathLengthInput = readLine() {
        let pathLength = Int(pathLengthInput) ?? 0

        if let pathInput = readLine() {
            let path = pathInput.components(separatedBy: " ").compactMap { Int($0) }

            checkPathValidity(n: n, m: m, corridors: corridors, pathLength: pathLength, path: path)
        }
    }
}
```

Результат работы программы 1:

```
swift /tmp/gKhyUbea20.swift
3 2
1 2 10
1 3 5
5
10 10 10 10 5

3
```

Результат работы программы 2:

```
swift /tmp/rQynyxrLuS.swift
3 2
1 2 10
2 3 5
5
5 10 10 10 10

INCORRECT
```

Результат работы программы 3:

```
swift /tmp/md3Qypq5Ft.swift
3 2
1 2 10
1 3 5
4
10 10 10 5

INCORRECT
```

Задача №13: Грядки (3 балл)

Прямоугольный садовый участок шириной N и длиной M метров разбит на квадраты со стороной 1 метр. На этом участке вскопаны грядки. Грядкой называется совокупность квадратов, удовлетворяющая таким условиям:

- из любого квадрата этой грядки можно попасть в любой другой квадрат этой же грядки, последовательно переходя по грядке из квадрата в квадрат через их общую сторону;
- никакие две грядки не пересекаются и не касаются друг друга ни по вертикальной, ни по горизонтальной сторонам квадратов (касание грядок углами квадратов допускается).

Подсчитайте количество грядок на садовом участке.

Формат входных данных (input.txt) и ограничения.
В первой строке входного файла INPUT.TXT находятся числа N и M через пробел, далее идут N строк по M символов. Символ # обозначает территорию грядки, точка соответствует незанятой территории. Других символов в исходном файле нет ($1 \le N, M \le 200$).

Формат выходных данных (output.txt).
В выходной файл OUTPUT.TXT выведите количество грядок на садовом участке.

Код программы:

```swift
import Foundation

// Read input from console
func readInput() -> (Int, Int, [[Character]]) {
    guard let input = readLine()?.split(separator: " ") else {
        fatalError("Invalid input")
    }

    let N = Int(input[0])!
    let M = Int(input[1])!

    var garden: [[Character]] = []
    for _ in 0..<N {
```

```
        guard let row = readLine()?.map({$0}) else {
            fatalError("Invalid input")
        }
        garden.append(row)
    }

    return (N, M, garden)
}

// Count the number of beds in the garden
func countBeds(N: Int, M: Int, garden: [[Character]]) -> Int {
    var visited = Array(repeating: Array(repeating: false, count: M), count: N)
    var bedsCount = 0

    func dfs(_ i: Int, _ j: Int) {
        if i < 0 || i >= N || j < 0 || j >= M || garden[i][j] == "." || visited[i][j] {
            return
        }

        visited[i][j] = true

        // Check all four surrounding cells
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)
    }

    for i in 0..<N {
        for j in 0..<M {
            if garden[i][j] == "#" && !visited[i][j] {
                bedsCount += 1
                dfs(i, j)
            }
        }
    }

    return bedsCount
}

func main() {
    let (N, M, garden) = readInput()

    let startTime = Date()

    let bedsCount = countBeds(N: N, M: M, garden: garden)

    let endTime = Date()
    let timeElapsed = endTime.timeIntervalSince(startTime)

    // Print the number of beds and the time taken
    print("")
    print("Количество грядок: \(bedsCount)")
```

```
    print("Время работы алгоритма: \(timeElapsed) секунд")
}

// Run the program
main()
```

Результат работы программы 1:

```
swift /tmp/KrJSsehAEI.swift
5 10
##......#.
.#..#...#.
.###....#.
..##....#.
........#.

Количество грядок: 3
Время работы алгоритма: 5.4001808166503906e-05 секунд
```

Результат работы программы 2:

```
swift /tmp/N6kVuEbTLb.swift
5 10
##..#####.
.#.#.#....
###..##.#.
..##.....#
.###.#####

Количество грядок: 5
Время работы алгоритма: 3.898143768310547e-05 секунд
```

Задача №14: Автобусы (3 балл)

Между некоторыми деревнями края Власюки ходят автобусы. Поскольку пассажиропотоки здесь не очень большие, то автобусы ходят всего несколько раз в день.
Марии Ивановне требуется добраться из деревни d в деревню v как можно быстрее (считается, что в момент времени 0 она находится в деревне d).

Формат входных данных (input.txt) и ограничения.
Во входном файле INPUT.TXT записано число N – общее число деревень ($1 \le N \le 100$),

номера деревень d и v, затем количество автобусных рейсов R ($0 \le R \le 10000$). Затем идут описания автобусных рейсов. Каждый рейс задается номером деревни отправления, временем отправления, деревней назначения и временем прибытия (все времена - целые от 0 до 10000). Если в момент t пассажир приезжает в деревню, то уехать из нее он может в любой момент времени, начиная с t.

Формат выходных данных (output.txt).
В выходной файл OUTPUT.TXT вывести минимальное время, когда Мария Ивановна может оказаться в деревне v. Если она не сможет с помощью указанных автобусных рейсов добраться из d в v, вывести -1.

Код программы:

```
import Foundation

// Чтение входных данных из консоли
let input = readLine()!.split(separator: " ").map { Int($0)! }
let n = input[0]
let d = input[1]
let v = input[2]
let r = input[3]

// Инициализация переменной, представляющей минимальное время для прибытия в деревню v
var minArrivalTime = Int.max

// Считываем описания автобусных рейсов
for _ in 0..<r {
    let route = readLine()!.split(separator: " ").map { Int($0)! }
    // Проверяем, является ли этот рейс подходящим для Марии Ивановны
    if route[0] == d && route[2] == v {
        // Обновляем минимальное время прибытия
        minArrivalTime = min(minArrivalTime, route[3])
    }
}

// Вывод результата в консоль
if minArrivalTime == Int.max {
    print("-1")
} else {
    print(minArrivalTime)
}
```

Результат работы программы:

```
swift /tmp/58CMKtocbd.swift
3 1 3 4
1 0 2 5
1 1 2 3
2 3 3 5
1 1 3 10
5
```

Задача №16: Рекурсия (3 балл)

Одним из важных понятий, используемых в теории алгоритмов, является рекурсия. Неформально ее можно определить как использование в описании объекта самого себя. Если речь идет о процедуре, то в процессе исполнении эта процедура напрямую или косвенно (через другие процедуры) вызывает сама себя.
Рекурсия является очень «мощным» методом построения алгоритмов, но таит в себе некоторые опасности. Например, неаккуратно написанная рекурсивная процедура может войти в бесконечную рекурсию, то есть, никогда незакончить свое выполнение (на самом деле, выполнение закончится с переполнением стека).
Поскольку рекурсия может быть косвенной (процедура вызывает сама себя через другие процедуры), то задача определения того факта, является ли данная процедура рекурсивной, достаточно сложна. Попробуем решить более простую задачу.
Рассмотрим программу, состоящую из n процедур P1, P2, . . . , Pn. Пусть для каждой процедуры известны процедуры, которые она может вызывать. Процедура P называется потенциально рекурсивной, если существует такая последовательность процедур Q0, Q1, . . . , Qk, что Q0 = Qk = P и для i = 1...k процедура Qi−1 может вызвать процедуру Qi. В этом случае задача будет заключаться в определении для каждой из заданных процедур, является ли она потенциально рекурсивной.
Требуется написать программу, которая позволит решить названную задачу.

Формат входных данных (input.txt) и ограничения.
Первая строка входного файла INPUT.TXT содержит целое число n – количество процедур в программе ($1 \le n \le 100$). Далее следуют n блоков, описывающих процедуры.
После каждого блока следует строка, которая содержит 5 символов «*».
Описание процедуры начинается со строки, содержащий ее идентификатор, состоящий только из маленьких букв английского алфавита и цифр. Идентификатор непуст, и его длина не превосходит 100 символов. Далее идет строка, содержащая число k ($k \le n$) – количество процедур, которые могут быть вызваны описываемой процедурой. Последующие k строк содержат идентификаторы этих процедур – по одному идентификатору на строке.
Различные процедуры имеют различные идентификаторы. При этом ни одна процедура не может вызвать процедуру, которая не описана во входном файле.

Формат выходных данных (output.txt).

В выходной файл OUTPUT.TXT для каждой процедуры, присутствующей во входных данных, необходимо вывести слово YES, если она является потенциально рекурсивной, и слово NO – в противном случае, в том же порядке, в каком они перечислены во входных данных.

Код программы:

```
import Foundation

var recursionDictionary: [String: Bool] = [:]
var procedures: [String] = []

func isPotentiallyRecursive(_ procedure: String, _ graph: inout [String: [String]]) -> Bool {
    if recursionDictionary.keys.contains(procedure) {
        return recursionDictionary[procedure] ?? false
    }

    if let proceduresToCall = graph[procedure] {
        for proc in proceduresToCall {
            if proc == procedures[0] {
                recursionDictionary[procedure] = true
                return true
            } else {
                if isPotentiallyRecursive(proc, &graph) {
                    recursionDictionary[procedure] = true
                    return true
                }
            }
        }
    }

    recursionDictionary[procedure] = false
    return false
}

func checkRecursion(_ numOfProcedures: Int, _ procedureDetails: [[String]]) {
    var graph: [String: [String]] = [:]

    for detail in procedureDetails {
        let currentProcedure = detail[0]
        let numCalls = Int(detail[1]) ?? 0
        var calls: [String] = []

        for i in 2...numCalls + 1 {
            calls.append(detail[i])
        }

        graph[currentProcedure] = calls
    }

    for procedure in procedures {
        let result = isPotentiallyRecursive(procedure, &graph)
        if result {
```

```
            print("YES")
        } else {
            print("NO")
        }
    }
}

if let input = readLine() {
    if let numberOfProcedures = Int(input) {
        for _ in 0..<numberOfProcedures {
            if let procedureID = readLine(), let numToCall = readLine() {
                let num = Int(numToCall) ?? 0
                var procedureDetails: [String] = [procedureID, numToCall]
                for _ in 0..<num {
                    if let proc = readLine() {
                        procedureDetails.append(proc)
                    }
                }
                procedures.append(procedureID)
                checkRecursion(numberOfProcedures, [procedureDetails])
            }
        }
    }
}
```

Результат работы программы:

```
swift /tmp/Hh4pNereUI.swift
3
p1
2
p1
p2
YES
p2
1
p1
YES
YES
p3
1
p1
YES
YES
NO
```

Вывод:

Решил задачи: 1(1б), 2(1б), 3(1б), 4(1б), 5(1.5б), 6(1б), 7(1.5б), 12(2б), 13(3б), 14(3б), 16(3б). В сумме = 19 баллов