МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Поиск с возвратом

Студент гр. 3343 _____ Отмахов Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Изучение алгоритма поиска с возвратом, реализация с его помощью программы, решающей задачу размещения квадратов на столе.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до $N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу – квадрат размера N. Он может получить ее, собрав из уже имеющихся обрезков (квадратов). Например, столешница размера 7×7 может быть построена из 9 обрезков

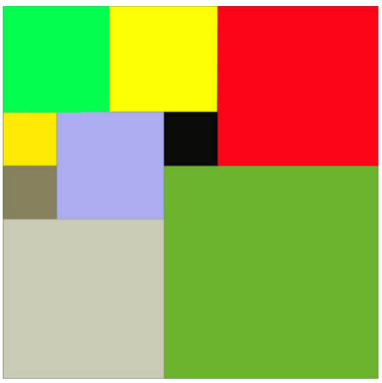

Рисунок 1 – пример размещения квадратов

Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N $(2 \le N \le 20)$.

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x,y и w, задающие координаты левого верхнего угла $(1 \le x,y \le N)$ и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Вар. 2и. Итеративный бэктрекинг. Исследование времени от размера квадрата.

Основные теоретические положения.

Поиск с возвратом (backtracking) — это общий метод поиска решений задач, которые требуют полного перебора всех возможных вариантов из некоторого множества. Этот метод заключается в том, что решение задачи строится постепенно, шаг за шагом, начиная с частичного решения. Если на каком-то шаге расширить текущее решение не удается (например, оно не удовлетворяет ограничениям задачи), то происходит возврат к предыдущему шагу, и из более короткого частичного решения продолжается поиск.

Алгоритм поиска с возвратом позволяет найти все возможные решения задачи, если они существуют, и является особенно полезным в случаях, когда решение невозможно найти простым перебором всех вариантов сразу. Вместо

этого, с помощью откатов, можно исключать неверные варианты и сосредотачиваться на тех, которые могут привести к решению.

Выполнение работы.

Для решения задачи был использован итеративный поиск с возвратом. Алгоритм пытается разместить квадратные обрезки на доске, начиная с самого большого возможного размера. Для каждого размера квадрата проверяется, можно ли его разместить в текущей пустой области доски. Если размещение невозможно, алгоритм откатывается и пробует другой вариант. Этот процесс продолжается до тех пор, пока столешница не будет полностью покрыта.

Алгоритм использует очередь для хранения частичных решений и итеративно расширяет решение, добавляя новые квадраты. Если не удается продолжить решение, происходит откат. Как только столешница заполняется, алгоритм завершает выполнение. В результате выводится минимальное количество квадратов и их координаты на доске.

Описание методов и структур данных:

- Структура *Square* – используется для представления квадратных обрезков, которые размещаются на столешнице. Она содержит:
  - *size* - целое число, представляющее размер квадрата (длину его стороны).
  - *x* - координата X (левая верхняя точка) квадрата на доске.
  - *x* - координата Y (левая верхняя точка) квадрата на доске.
- Класс *Desk* – основная структура, представляющая столешницу, которую необходимо покрыть квадратами. Она содержит:
  - *size* – размер столешницы.
  - *map* – двумерный массив (матрица), представляющий столешницу. Каждая ячейка матрицы содержит значение, указывающее на

принадлежность этой ячейки какому-либо квадрату. Если ячейка пустая, то значение равно 0.

- *count* – количество использованных квадратных обрезков.
- *squares* – массив, содержащий все размещенные квадраты (их размер и координаты).
- *emptyCell* – опциональный кортеж, представляющий координаты первой пустой ячейки (если такая существует), или *nil*, если столешница полностью покрыта.
- *init(size: Int)* – инициализатор, создающий столешницу размером N×N и инициализирующий все остальные параметры.
- *setDefault()* – метод, создающий начальную раскладку для столешницы (в случае нечётных размеров).
- *setOptimizedEven()* – метод оптимизации для четных n.
- *setOptimizedPrime()* – метод оптимизации для простых n.
- *isPrime(_ n: Int)* → *Bool* – проверяет простое ли число n.
- *updateEmptyCell()* – обновляет координаты первой пустой ячейки на столешнице.
- *isFull()* → *Bool* – проверяет, заполнена ли столешница полностью (нет *пустых ячеек).*
- *addSquare(size: Int, x: Int, y: Int)* – добавляет квадрат на столешницу в заданные координаты и обновляет все необходимые данные.
- *canAdd(size: Int, x: Int, y: Int)* – проверяет, можно ли разместить квадрат заданного размера на доске в координатах (x, y).
- *printDesk()* – выводит текущее состояние столешницы в консоль для визуализации.

- Метод *backtracking(desk: Desk)* – реализует итеративный поиск с возвратом. Он пытается заполнить столешницу с помощью квадратов, используя очередь для хранения частичных решений.

Оценка временной сложности:

Основная часть алгоритма — это поиск в ширину (BFS) по возможным расстановкам квадратов:

1. Перебор пустых ячеек:
   - В каждой итерации алгоритма мы выбираем первую свободную клетку (за $O(n^2)$ в худшем случае).
2. Перебор возможных квадратов:
   - Мы перебираем размеры квадратов от $\mathrm{n} - 1$ до 1 (всего вариантов $O(n)$).
   - Для каждого размера проверяется, можно ли его разместить, что требует проверки всех клеток в этом квадрате: $O(s^2)$, где s – текущий размер квадрата. В худшем случае это $O(n^2)$.
3. Количество состояний (размер очереди в BFS):
   - Максимальная глубина рекурсивного вызова связана с минимальным количеством квадратов, покрывающих область. Это ограничивается $O(\log n)$.
   - Однако, в общем случае, BFS может обойти экспоненциальное количество состояний (каждое новое состояние вносит новую расстановку квадратов).

Таким образом, оценка времени выполнения сильно зависит от конкретных случаев. В худшем случае алгоритм экспоненциальный: $O(n^2)$.

Оценка сложности алгоритма по памяти:

$O(n^2)$ – на создание поля n x n и хранение лучшего и текущего разбиений.

Код программы представлен в Приложении А.

Исследование.

Для анализа работы алгоритма разбиения квадрата на минимальное количество меньших квадратов был проведен эксперимент, целью которого было выяснить, как изменяется время выполнения программы в зависимости от размера квадрата *n*.

Этапы исследования:

1. Программа запускалась для значений *n* от 2 до 20.
2. Замер времени выполнения проводился с помощью *CFAbsoluteTimeGetCurrent()*, фиксируя момент начала и конца работы алгоритма.
3. Для каждого размера квадрата программа выполнялась один раз, и полученные результаты заносились в таблицу.

Таблица 1 – Результаты исследования

Размер квадрата	**Время выполнения, сек**
2	0.00004804134368896484
3	0.00008392333984375
4	0.00025033950805664062
5	0.0004320144653320312
6	0.0004494190216064453
7	0.001804947853088379
8	0.0007104873657226562
9	0.012501001358032227
10	0.0010597705841064453
11	0.03858804702758789
12	0.0014209747314453125
13	0.09305500984191895
14	0.0019204616546630859
15	0.15693306922912598
16	0.0024604797363281250

17	0.790789008140564
18	0.00029397010803222656
19	2.7583431005477905
20	0.00037109851837158203

4. На основе данных таблицы был построен график зависимости времени работы от размера квадрата.

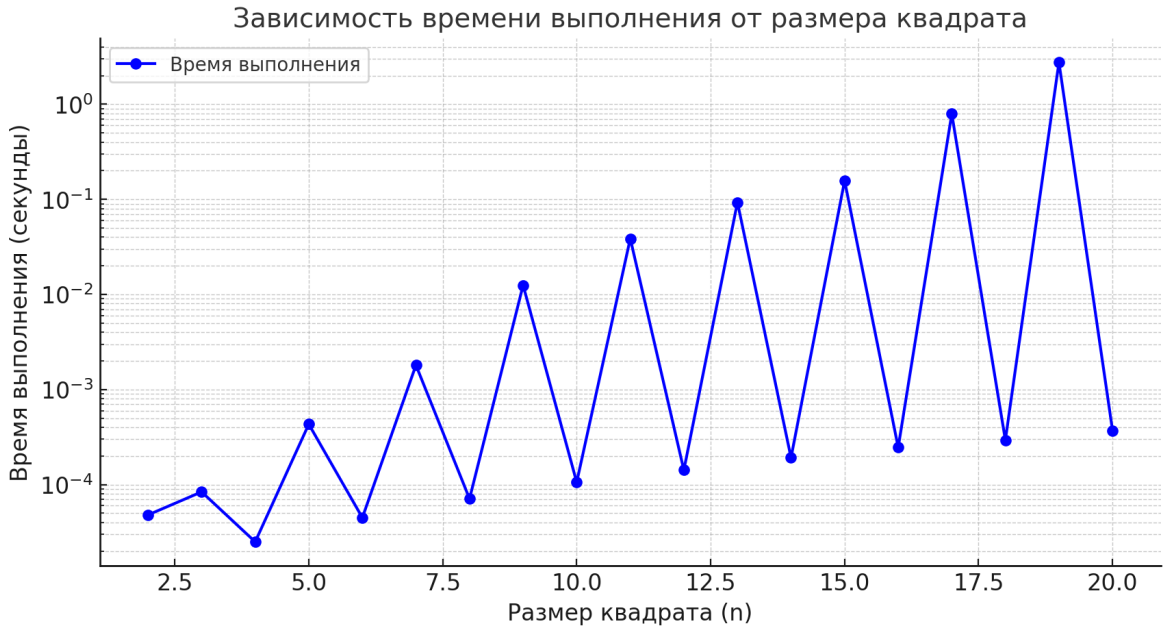

Рисунок 1 – Зависимость времени выполнения от размера квадрата

График показывает нелинейный рост времени выполнения с увеличением размера квадрата. На малых значениях время колеблется из-за возможных системных факторов, но на больших размерах наблюдается резкий экспоненциальный рост, что указывает на сложность алгоритм.

Пример работы программы.

```
Enter the size of the square:
7
9
0 0 4
0 4 3
4 0 3
3 4 2
3 6 1
4 3 1
4 6 1
5 3 2
5 5 2
Program ended with exit code: 0
```

Рисунок 2 – Пример работы программы

Тестирование.

Таблица 2 – Результаты тестирования

Ввод	**Вывод**	**Ожидаемый результат**
2	4 0 0 1 0 1 1 1 0 1 1 1 1	Результат верный
3	6 0 0 2 0 2 1 1 2 1 2 0 1 2 1 1 2 2 1	Результат верный
4	4 0 0 2 0 2 2 2 0 2 2 2 2	Результат верный

5	8 0 0 3 0 3 2 3 0 2 2 3 2 3 2 1 4 2 1 4 3 1 4 4 1	Результат верный
6	4 0 0 3 0 3 3 3 0 3 3 3 3	Результат верный
7	9 0 0 4 0 4 3 4 0 3 3 4 2 3 6 1 4 3 1 4 6 1 5 3 2 5 5 2	Результат верный
8	4 0 0 4 0 4 4 4 0 4 4 4 4	Результат верный
9	6 0 0 6 0 6 3 3 6 3 6 0 3 6 3 3 6 6 3	Результат верный
10	4 0 0 5 0 5 5 5 0 5 5 5 5	Результат верный
1	Error: The size of the square should be in the range from 2 to 20.	Результат верный

21	Error: The size of the square should be in the range from 2 to 20.	Результат верный

Выводы.

В ходе выполнения лабораторной работы было разработано решение задачи разбиения квадрата при помощи поиска с возвратом, а также проведено исследование зависимости времени работы алгоритма от размера квадрата.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.swift

```
//
//  main.swift
//  Backtracking
//
//  Created by Danil Otmakhov on 16.02.2025.
//

import Foundation

enum InputError: Error {
    case outOfRange
}

struct Square {
    let size: Int
    let x: Int
    let y: Int
}

class Desk {

    let size: Int
    var map: [[Int]]
    var count: Int
    var squares: [Square]
    var emptyCell: (Int, Int)?

    init(_ size: Int) {
        self.size = size
        self.map = Array(repeating: Array(repeating: 0, count: size), count:
size)
        self.squares = []
        self.count = 0
        self.emptyCell = (0, 0)

        if size % 2 == 0 {
            setOptimizedEven()
        } else if isPrime(size) {
            setOptimizedPrime()
        } else if size % 2 != 0 && size % 3 != 0 {
            self.setDefault()
        }
    }

    private func setDefault() {
        addSquare(size: (size + 1) / 2, x: 0, y: 0)
        addSquare(size: (size - 1) / 2, x: 0, y: (size + 1) / 2)
        addSquare(size: (size - 1) / 2, x: (size + 1) / 2, y: 0)
    }

    private func setOptimizedEven() {
        let half = size / 2
        for i in 0..<2 {
            for j in 0..<2 {
```

```
            addSquare(size: half, x: i * half, y: j * half)
        }
    }
}

private func setOptimizedPrime() {
    addSquare(size: (size + 1) / 2, x: 0, y: 0)
    addSquare(size: size / 2, x: 0, y: (size + 1) / 2)
    addSquare(size: size / 2, x: (size + 1) / 2, y: 0)
}

private func isPrime(_ n: Int) -> Bool {
    if n < 2 { return false }
    if n < 4 { return true }
    for i in 2...Int(sqrt(Double(n))) {
        if n % i == 0 {
            return false
        }
    }
    return true
}

private func updateEmptyCell() {
    for i in 0..<size {
        for j in 0..<size {
            if map[i][j] == 0 {
                emptyCell = (i, j)
                return
            }
        }
    }

    emptyCell = nil
}

func isFull() -> Bool {
    return emptyCell == nil
}

func addSquare(size: Int, x: Int, y: Int) {
    for i in x..<x + size {
        for j in y..<y + size {
            map[i][j] = count + 1
        }
    }

    let square = Square(size: size, x: x, y: y)
    squares.append(square)
    count += 1

    updateEmptyCell()
}

func canAdd(size: Int, x: Int, y: Int) -> Bool {
    if y + size > self.size || x + size > self.size {
        return false
    }

    for i in x..<x + size {
        for j in y..<y + size {
            if map[i][j] != 0 {
```

```
                    return false
                }
            }
        }

        return true
    }

    func printDesk() {
        var output = Array(repeating: Array(repeating: 0, count: size), count:
size)

        for square in squares {
            for i in square.x..<square.x + square.size {
                for j in square.y..<square.y + square.size {
                    output[i][j] = square.size
                }
            }
        }

        for row in output {
            print(row.map { String($0) }.joined(separator: " "))
        }
    }
}

func backtracking(desk: Desk) -> Desk {
    var queue: [Desk] = [desk]

    while let first = queue.first, let emptyCell = first.emptyCell {
        queue.removeFirst()

        for i in (1..<first.size).reversed() {
            if first.canAdd(size: i, x: emptyCell.0, y: emptyCell.1) {
                let current = Desk(first.size)
                current.map = first.map
                current.squares = first.squares
                current.count = first.count

                current.addSquare(size: i, x: emptyCell.0, y: emptyCell.1)

                print("\nStep: \(current.count). Added square at
(\(emptyCell.0), \(emptyCell.1)) with size \(i)")
                current.printDesk()

                if current.isFull() {
                    return current
                }

                queue.append(current)
            }
        }
    }

    return queue.first ?? desk
}

do {
    print("Enter the size of the square:")
    let n = Int(readLine()!)!
    guard n >= 2, n <= 20 else {
```

```
        throw InputError.outOfRange
    }
    let start = CFAbsoluteTimeGetCurrent()

    let desk = Desk(n)
    let answer = backtracking(desk: desk)

    let end = CFAbsoluteTimeGetCurrent()
    print("\nTime to complete: \(end - start) seconds")
    print(answer.count)
    for square in answer.squares {
        print("\(square.x) \(square.y) \(square.size)")
    }
} catch InputError.outOfRange {
    print("Error: The size of the square should be in the range from 2 to 20.")
}
```