# 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 |  | Атоян М.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург 2025

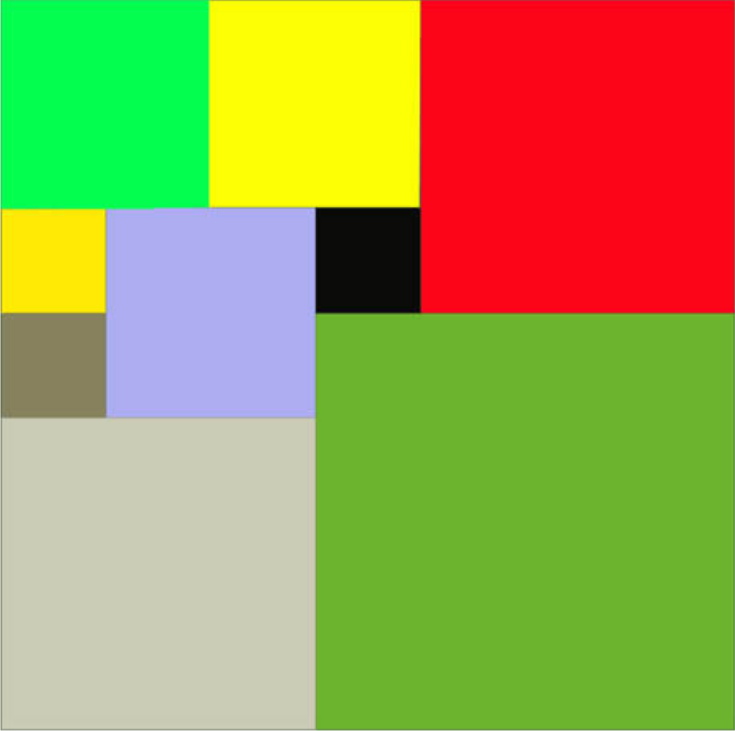
# Цель работы.

Решение классической задачи квадрирования квадрата (с заданными относительно размера ограничениями) посредством программы, основанной на алгоритме поиска с возвратом (англ. backtracking).

# Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменя- ются от 1 до *N*−1, и у него есть неограниченное число обрезков любого раз- мера. Но ему очень хочется получить большую столешницу - квадрат раз- мера *N*. Он может получить ее, собрав из уже имеющихся обрезков(квадра- тов).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



Внутри столешницы не должно быть пустот, обрезки не должны выходить за

пределы столешницы и не должны перекрываться. Кроме того, Вова хочет ис- пользовать минимально возможное число обрезков.

# Входные данные

Размер столешницы - одно целое число *N* (2 ≤ *N* ≤ 40).

# Выходные данные

Одно число *K*, задающее минимальное количество обрезков(квадратов), из ко- торых 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 ≤ *x*, *y* ≤ *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

**Вариант 2р.** Рекурсивный бэктрекинг. Исследование времени выполнения от

размера квадрата

# Описание функций и структур данных.

*type Square struct* - структура, задающая квадрат. Полями являются целочисленные значения. Поля структуры:

* *X, Y* – координаты левого верхнего угла квадрата.
* *Size* – длина стороны квадрата.

*type Result struct* - cтруктура для хранения результата. Поля структуры:

* *Count* – количество квадратов в итоговом разбиении.
* *Squares* – массив квадратов, представляющих итоговое разбиение.

*type Table struct* - основная структура, представляющая таблицу и состояние решения. Поля структуры:

* *N* – размер исходного квадрата (таблицы), который нужно покрыть меньшими квадратами.
* *matrix* – двумерный массив (матрица), представляющий карту размещения квадратов. Каждая ячейка содержит размер квадрата, который её занимает, или 0, если ячейка свободна.
* *current* – массив квадратов, представляющий текущее промежуточное разбиение.
* *result* – массив квадратов, представляющий лучшее найденное разбиение.
* *bestCount* – минимальное количество квадратов в лучшем разбиении.
* *currentCount* – количество квадратов в текущем промежуточном разбиении.
* *maxSquareSize* – максимальный размер квадрата, который можно разместить на текущем этапе.

Методы структуры Table:

1. *New(n int) \*Table:* Создает и возвращает новый объект Table для квадрата размера n x n. Инициализирует матрицу и устанавливает начальные значения для переменных.
2. *Place(x, y, size int) error:* Пытается разместить квадрат размером size с координатами (x, y) на таблице. Проверяет, что квадрат не выходит за границы таблицы и не пересекается с другими квадратами. Если размещение возможно, заполняет соответствующие ячейки матрицы значением size. В случае ошибки возвращает сообщение.
3. *FindEmptyX(y int) int:* Ищет первую свободную ячейку в строке y. Возвращает координату x первой свободной ячейки или -1, если строка полностью заполнена.
4. *RemoveSquare(x, y, size int):* Удаляет квадрат размером size с координатами (x, y) из таблицы, освобождая соответствующие ячейки матрицы.
5. *Backtrack(y int):* Рекурсивная функция, реализующая алгоритм поиска с возвратом (backtracking). Начиная с строки y, пытается разместить квадраты, перебирая возможные размеры. Если текущее количество квадратов превышает лучшее найденное решение, прекращает дальнейший поиск. При успешном размещении всех квадратов обновляет лучшее решение.
6. *Optimize() error:* Пытается оптимизировать решение для таблиц с четным размером или с N = 2\*\*r-1. Если N четное, таблица делится на 4 квадрата размером N/2 x N/2. Если N = 2\*\*r-1 и притом простое, установлена закономеронсть, в оптимальном разбиении присутствуют: один квадрат длины 2\*\*(r-1), два квадрата длины 2\*\*r-1, три квадрата длины 2\*\*(r-2), три квадрата длины 2\*\*(r-3), и т.д. по тройкам квадратов. В заданных по условию границах n, оптимизация актуальна для n = 7. В таких случаях можем так же сразу получить ответ.
7. *Solve() Result:* Основная функция, которая запускает процесс решения задачи. Если оптимизация невозможна, использует backtracking для поиска минимального количества квадратов. Возвращает результат в виде структуры Result.

# Оптимизации:

1. При четности длины стороны квадрата – оптимальным решением представляется разбиение на 4 равных квадрата. Разбиение всегда допустимо за счет наличия двойки среди делителей числа (площадь квадрата предполагает ее возведение в степень 2 и соответственно делимость на 4). В таком случае, сразу получаем ответ.
2. При простом n установлены расположение и размер трех входящих в оптимальное решение квадратов: Один длиной (n+1)/2 и два (n+1)/2-1. Соответствующая тройка сразу включается в оптимальное разбиение.
3. Если N = 2\*\*r-1 и притом простое, установлена закономеронсть, в оптимальном разбиении присутствуют: один квадрат длины 2\*\*(r-1), два квадрата длины 2\*\*r-1, три квадрата длины 2\*\*(r-2), три квадрата длины 2\*\*(r-3), и т.д. по тройкам квадратов. В заданных по условию границах n, оптимизация актуальна для n = 7.
4. Если количество квадратов на текущем шаге больше или равно *t.bestCount*, то мы принудительно выходим из этой ветки рекурсии. В таком случае, мы избегаем излишних итераций по квадрату, если заведомо известно, что результат будет хуже имеющегося.

# Описание программы.

Происходит считывание числа n – длины квадрата. Далее вызывается вышеописанная функция *Solve()*, проверяющая условия оптимизаций. *Solve()* влечет за собой вызов функции *Backtrack (y int)*.

*Backtrack (y int)* – рекурсивная функция поиска с возвратом (backtracking). Она принимает на вход номер строки y, с которой начинается поиск места для размещения квадрата. Основная цель функции – найти оптимальное разбиение таблицы на квадраты.

**Шаги работы функции:**

1. **Проверка завершения:**
   * Если текущая строка y превышает размер таблицы (y >= t.N), это означает, что все строки успешно заполнены. В этом случае:
     + Проверяется, является ли текущее количество квадратов *t.currentCount* меньше лучшего найденного решения *t.bestCount*.
     + Если да, то текущее решение сохраняется как лучшее:

*t.bestCount = t.currentCount*

*t.result = make([]Square, t.bestCount)*

*copy(t.result, t.current)*

* + - Функция завершает выполнение для данной ветви рекурсии.

1. **Поиск свободной ячейки:**
   * С помощью функции *FindEmptyX(y int)* находится первая свободная ячейка в строке y. Если строка полностью заполнена (x == -1), функция переходит к следующей строке, вызывая *Backtrack(y + 1)*.
2. **Проверка целесообразности продолжения:**
   * Если текущее количество квадратов *t.currentCount* уже превышает лучшее найденное решение *t.bestCount*, дальнейший поиск прекращается. Это позволяет отсечь заведомо неоптимальные ветви.
3. **Определение максимального размера квадрата:**
   * Вычисляется максимальный размер квадрата, который можно разместить в текущей ячейке. Этот размер ограничен:
     + Размером таблицы *t.N - x*и *t.N - y*.
     + Максимальным допустимым размером квадрата *t.maxSquareSize*.
   * Переменная *size* принимает значение максимального допустимого размера.
4. **Перебор возможных размеров квадратов:**
   * В цикле перебираются все возможные размеры квадратов от size до 1:

*for size := maxSize; size >= 1; size-- {*

* + Для каждого размера:
    - Пытаемся разместить квадрат с помощью функции *Place(x, y, size int).* Если размещение успешно:
      * Квадрат добавляется в текущее решение *t.current*.
      * Увеличивается счетчик квадратов *t.currentCount*.
      * Вызывается *Backtrack(y)* для продолжения поиска в текущей строке.
    - Если размещение невозможно, переходим к следующему размеру.

1. **Возврат (Backtracking):**
   * После завершения рекурсивного вызова выполняется возврат:
     + Квадрат удаляется из текущего решения с помощью функции *RemoveSquare(x, y, size int)*.
     + Счетчик квадратов *t.currentCount* уменьшается .
     + Квадрат удаляется из массива текущих решений *t.current = t.current[:len(t.current)-1]*.
2. **Обновление максимального размера квадрата:**
   * Если произошел возврат до начального состояния (например, все квадраты удалены), максимальный размер квадрата снова устанавливается равным *t.N - 1*.

**Сложность алгоритма** по памяти составляет O(n^2)+O(log(n)) – затраты на матрицу-карту, хранящую решения и хранящие промежуточный и конечный ответ структуры.

**Сложность алгоритма** по операциям зависит от входных данных и варьируется от О(1) в случае четных чисел, O(log(N)) для N, которые можно представить как 2\*\*r-1, до экспоненциальной сложности в худшем случае, так как количество операций становится гораздо больше.

# Исследование.

# С помощью функции *BenchmarkSolve(b \*testing.B)* замерено время выполнения программы для каждого размера ребра квадрата в диапазоне от 2 до 20.

# Благодаря оптимизациям алгоритм крайне эффективно справляется с чётными числами и числом 7 = 2\*\*3-1. В остальных случаях наблюдается повышенное время исполнения, так как приходится делать полный перебор по площади квадрата.

# Тестирование.

|  |  |  |
| --- | --- | --- |
| Входные данные | Выходные данные | Комментарий |
| 7 | 9  1 1 4  1 5 3  5 1 3  6 4 2  4 6 2  6 6 2  5 4 1  4 5 1  5 5 1 | Оптимизация 3)  Результат верный |
| 15 | 12  1 1 8  1 9 7  9 1 7  12 8 4  8 12 4  12 12 4  10 8 2  8 10 2  10 10 2  9 8 1  8 9 1  9 9 1 | Оптимизация 4)  Результат верный |
| 16 | 4  1 1 8  1 9 8  9 1 8  9 9 8 | Оптимизация 1)  Результат верный |
| 19 | 13  1 1 10  1 11 9  11 1 9  11 10 3  14 10 6  10 11 1  10 12 1  10 13 4  14 16 1  15 16 1  16 16 4  10 17 3  13 17 3 | Оптимизация 2)  Результат верный |

# Выводы.

В соответствии с заданным условиям была написана программа, осуществляющая покрытие квадрата меньшими квадратами посредством поиска с возвратом. В ходе изучения поставленной задачи были выявлены и применены оптимизации, обеспечивающие значительное сокращение перебираемых решений.

Файл main.go

# ПРИЛОЖЕНИЕ А

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

package main

import (

"fmt"

)

func main() {

var n int

fmt.Scan(&n)

if n < 2 || n > 40 {

panic("Invalid size")

}

t := New(n)

r := t.Solve()

fmt.Print(r)

}

Файл main\_test.go

package main

import (

"fmt"

"testing"

)

func BenchmarkSolve(b \*testing.B) {

testCases := []struct {

size int

}{

{2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10},

{11}, {12}, {13}, {14}, {15}, {16}, {17}, {18}, {19}, {20},

}

for \_, tc := range testCases {

b.Run(fmt.Sprintf("Size:%d", tc.size), func(b \*testing.B) {

b.StopTimer()

t := New(tc.size)

b.StartTimer()

t.Solve()

})

}

}

Файл table.go

package main

import (

"fmt"

"math"

"time"

)

type Square struct {

X, Y, Size int

}

type Table struct {

N int

matrix [][]int

current []Square

result []Square

bestCount int

currentCount int

maxSquareSize int

}

type Result struct {

Count int

Squares []Square

TimeTaken time.Time

}

func (r Result) String() string {

res := fmt.Sprintf("%d\n", r.Count)

for \_, sq := range r.Squares {

res += fmt.Sprintf("%d %d %d\n", sq.X+1, sq.Y+1, sq.Size)

}

return res

}

func New(n int) \*Table {

t := &Table{

N: n,

bestCount: math.MaxInt32,

maxSquareSize: n - 1,

}

t.matrix = make([][]int, n)

for i := range t.matrix {

t.matrix[i] = make([]int, n)

}

return t

}

func (t \*Table) Place(x, y, size int) error {

if x+size > t.N || y+size > t.N {

return fmt.Errorf("Out of bounds placement")

}

for i := y; i < y+size; i++ {

for j := x; j < x+size; j++ {

if t.matrix[i][j] != 0 {

return fmt.Errorf("Could not place square at coordinates: x:%v y:%v", x, y)

}

}

}

for i := y; i < y+size; i++ {

for j := x; j < x+size; j++ {

t.matrix[i][j] = size

}

}

return nil

}

func (t \*Table) FindEmptyX(y int) int {

for x := 0; x < t.N; x++ {

if t.matrix[y][x] == 0 {

return x

}

}

return -1

}

func (t \*Table) RemoveSquare(x, y, size int) {

for i := y; i < y+size; i++ {

for j := x; j < x+size; j++ {

t.matrix[i][j] = 0

}

}

}

func (t \*Table) Backtrack(y int) {

if y >= t.N {

if t.currentCount < t.bestCount {

t.bestCount = t.currentCount

t.result = make([]Square, t.bestCount)

copy(t.result, t.current)

}

return

}

x := t.FindEmptyX(y)

if x == -1 {

t.Backtrack(y + 1)

return

}

if t.currentCount >= t.bestCount {

return

}

maxSize := min(t.maxSquareSize, t.N-x, t.N-y)

for size := maxSize; size >= 1; size-- {

if err := t.Place(x, y, size); err == nil {

t.current = append(t.current, Square{x, y, size})

t.currentCount++

t.Backtrack(y)

t.RemoveSquare(x, y, size)

t.current = t.current[:len(t.current)-1]

t.currentCount--

}

}

}

func (t \*Table) Optimize() error {

if t.N%2 == 0 {

t.result = []Square{

{0, 0, t.N / 2},

{t.N / 2, 0, t.N / 2},

{0, t.N / 2, t.N / 2},

{t.N / 2, t.N / 2, t.N / 2},

}

t.bestCount = 4

return nil

}

if isPowerOfTwoMinusOne(t.N) {

base := (t.N + 1) / 2

t.result = []Square{

{0, 0, base},

{0, base, base - 1},

{base, 0, base - 1},

}

t.bestCount = 3

squareSize := base / 2

indentation := squareSize

for squareSize > 0 {

t.result = append(t.result,

Square{t.N - indentation, t.N - squareSize - indentation, squareSize},

Square{t.N - squareSize - indentation, t.N - indentation, squareSize},

Square{t.N - indentation, t.N - indentation, squareSize},

)

t.bestCount += 3

squareSize /= 2

indentation += squareSize

}

return nil

}

return fmt.Errorf("Could not optimize calculations")

}

func (t \*Table) Solve() Result {

if err := t.Optimize(); err == nil {

result := Result{

Count: t.bestCount,

Squares: t.result,

}

return result

}

if isPrime(t.N) {

base := (t.N + 1) / 2

t.Place(0, 0, base)

t.Place(0, base, base-1)

t.Place(base, 0, base-1)

t.current = []Square{

{0, 0, base},

{0, base, base - 1},

{base, 0, base - 1},

}

t.currentCount = 3

t.maxSquareSize = base - 1

}

t.Backtrack(0)

result := Result{

Count: t.bestCount,

Squares: t.result,

}

return result

}

Файл utils.go

package main

func isPrime(n int) bool {

    if n <= 1 {

        return false

    }

    for i := 2; i\*i <= n; i++ {

        if n%i == 0 {

            return false

        }

    }

    return true

}

func min(values ...int) int {

    m := values[0]

    for \_, v := range values[1:] {

        if v < m {

            m = v

        }

    }

    return m

}

func isPowerOfTwoMinusOne(n int) bool {

    return (n+1)&n == 0

}

func getExponent(n int) int {

    r := 0

    for n > 1 {

        n >>= 1

        r++

    }

    return r

}