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

Вариант: 4р

Студент гр. 3388 _____ Потапов Р.Ю.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

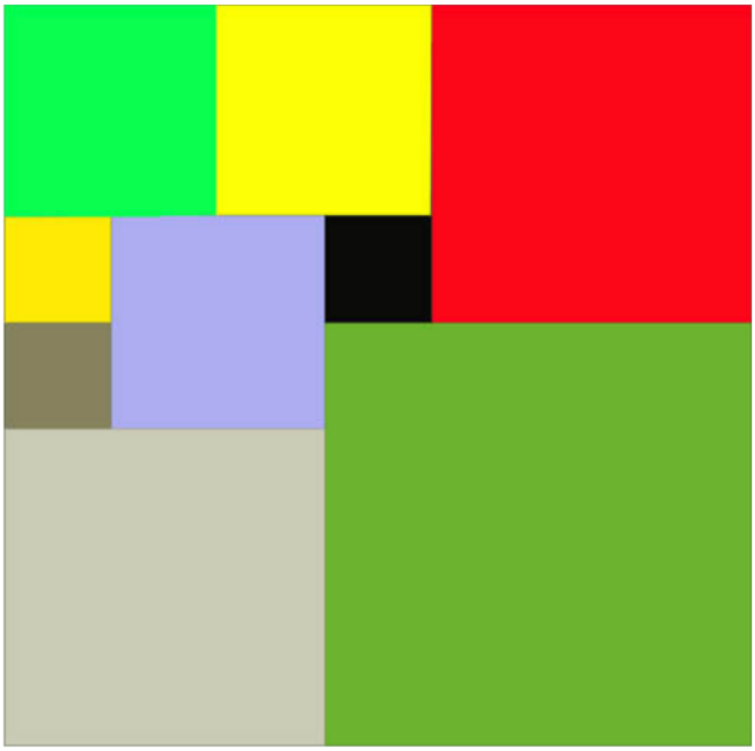
Цель работы:

Изучить теоретические основы алгоритма поиска с возвратом. Решить с его помощью задачу о разбиении квадрата. Провести исследование зависимости количества итераций от стороны квадрата.

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.



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

Выполнение работы

Описание алгоритма:

Рекурсивный алгоритм разбиения квадрата на минимальное количество подквадратов основан на методе backtracking (возврат к исходным данным). Основная идея алгоритма заключается в разбиении квадрата размером N×N на минимальное количество меньших квадратов, используя метод рекурсивного перебора с отсечениями (backtracking). Алгоритм стремится найти оптимальное покрытие квадрата, минимизируя количество используемых подквадратов.

Основные этапы работы алгоритма:

1. Масштабирование квадрата

Если размер квадрата N можно масштабировать (т.е. N имеет делители, отличные от 1 и самого себя), алгоритм уменьшает размер задачи, работая с меньшей сеткой.

Например, если N=6, его можно масштабировать до 3×3 с коэффициентом масштабирования 2. Это упрощает вычисления, так как задача решается для меньшей сетки, а результат затем масштабируется обратно.

2. Метод постановки трех начальных квадратов

Если масштабирование невозможно (например, N — простое число), алгоритм использует стратегию начального разбиения:

Размещает один большой квадрат размером (N+1)/2 в левом верхнем углу.

Размещает два меньших квадрата размером N/2 в оставшихся областях.

Это начальное разбиение помогает сократить пространство поиска и ускорить нахождение оптимального решения.

3. Рекурсивный перебор с отсечениями

Алгоритм рекурсивно перебирает все возможные варианты размещения квадратов, начиная с максимально возможного размера и уменьшая его до минимального.

Для каждой свободной клетки:

Определяется максимальный размер квадрата, который можно разместить в этой клетке без пересечения с уже занятыми областями.

Если квадрат успешно размещен, алгоритм продолжает поиск для оставшейся свободной области.

Если текущее количество квадратов превышает уже найденное оптимальное значение, алгоритм прекращает дальнейший перебор в этой ветке (отсечение).

4. Оптимизация через отсечения

Алгоритм отслеживает текущее количество квадратов и сравнивает его с лучшим найденным решением.

Если текущее решение уже хуже (использует больше квадратов), алгоритм прекращает дальнейший перебор в этой ветке, что значительно сокращает время выполнения.

5. Визуализация результата

После нахождения оптимального разбиения алгоритм визуализирует результат, создавая изображение, на котором каждый квадрат выделен своим цветом.

Это позволяет наглядно оценить, как квадрат был разбит на меньшие части.

Описание функций:

Функция solve_square_legacy(N)

• Проверяет, делится ли N на 2 или на 3 (быстрые случаи).

• Если N чётно, сразу возвращает разбиение на четыре квадрата со стороной N/2.

• Если N кратно 3, возвращает разбиение на шесть квадратов (2/3 N и 1/3 N).

• Если N не подходит под эти быстрые проверки, создаёт двумерный массив grid размером N \times N, который будет хранить, занята ли клетка (1) или свободна (0).

• Определяет «предварительные» три квадрата (pre), чтобы покрыть часть доски перед запуском основного перебора (большой квадрат (N+1)//2 и два поменьше). Ставит их в grid.

• Запускает функцию backtrack(0), которая ищет оптимальное покрытие всей доски.

• Итоговое решение состоит из трёх заранее поставленных квадратов и тех, что удалось найти в бэктрекинге.

3. Функция find_empty()

• Линейно пробегает массив grid сверху вниз, слева направо.

• Возвращает координаты (x, y) первой свободной клетки, в которой grid[y][x] == 0.

• Если свободных клеток нет, возвращает None, означая, что всё покрыто.

4. Функция can_place(x, y, s)

• Проверяет, можно ли поставить квадрат со стороной s в точке (x,y).

• Убеждается, что квадрат не выходит за границы (0 \le x + dx < N и 0 \le y + dy < N) и что все клетки в его области свободны (grid[y+dy][x+dx] == 0).

• Возвращает True, если разместить можно, иначе False.

5. Функция place(x, y, s, value)

• Проходит по области (x..x+s-1,\ y..y+s-1) и проставляет в grid[y+dy][x+dx] = value.

• При value=1 «занимает» ячейки квадрата, при value=0 — «освобождает».

6. Функция backtrack(count)

Рекурсивная функция бэктрекинга

• Сигнатура: def backtrack(count): ...

• Назначение: выполнить поиск оптимального покрытия оставшейся свободной части доски, добавляя новый квадрат за квадратом.

• Аргументы:

• count — текущее количество квадратов, уже поставленных на доску.

(Не возвращает отдельных значений, так как обновляет глобальную переменную best и добавляет/убирает элементы в список sol.)

• Возвращаемое значение: функция не возвращает результат напрямую. Вместо этого, когда доска полностью покрыта, она обновляет глобальную переменную best, хранящую лучшее решение (число квадратов и их список), а также оперирует списком sol, где отражается текущая расстановка.

При каждом вызове:

1. Если count >= best[0], происходит отсечение (выход из функции).

2. Если нет свободных клеток (find_empty() вернул None), сохраняем текущее покрытие как «лучшее».

3. Иначе находим первую пустую клетку, перебираем все возможные размеры квадрата (от максимального до 1). Для каждого подходящего варианта ставим квадрат (добавляем в sol), вызываем backtrack(count+1) и затем убираем квадрат (удаляем из sol), освобождая клетки.

7. Способ хранения частичных решений

Текущий набор уже расставленных квадратов хранится в списке sol.

• Каждый элемент в sol имеет вид (x+1, y+1, s), где (x, y) — координаты левого верхнего угла в сетке, а s — размер стороны.

• На каждом шаге бэктрекинга при установке нового квадрата соответствующий кортеж добавляется в sol, а при откате (pop) — убирается из него. Тем самым sol всегда отражает «текущую» частичную раскладку.

8. Визуализация

• В конце, уже после выполнения solve_tiling(N), программа строит график (matplotlib), на котором каждый найденный квадрат отображается в виде прямоугольника (контур).

• Ось x идёт слева направо, ось y — сверху вниз (за счёт ax.invert_yaxis()).

• Пределы устанавливаются от 1 до N + 1, чтобы соответствовать размеру исходного квадрата.

9. Использованные оптимизации:

Быстрые проверки для N \times N (если N делится на 2 или 3):

В случае чётного N или N, кратного 3, есть заранее известное разбиение квадрата на несколько крупных блоков. Алгоритм сразу возвращает готовое решение (4 или 6 квадратов) без запуска полного перебора.

• Предварительная расстановка трёх крупных квадратов:

Если N не подходит под «быстрые» случаи, часть доски изначально закрывается тремя большими квадратами. Это уменьшает зону, которую нужно покрыть перебором, и сокращает глубину рекурсии.

• Отсечения в бэктрекинге:

Когда текущее число уже поставленных квадратов \ge «лучшего» (минимального) найденного значения, дальнейшее углубление в данной ветке бессмысленно. Алгоритм «обрубает» ветку и переходит к следующей.

Оценка сложности алгоритма:

Временная сложность

• Худший случай: O(N^5).

Когда число N не делится на 2 или 3, алгоритм запускает полный бэктрекинг. В нём:

1. Ищется первая свободная клетка (из возможных N^2 клеток).

2. Для неё перебирается до N потенциальных размеров квадрата.

3. Каждая проверка «можно ли поставить квадрат» (функция can_place) потенциально пробегает до O(N^2) ячеек, если квадрат максимального размера.

Объединив эти факторы, в самом неблагоприятном сценарии получаем O(N^2) * O(N) * O(N^2) = O(N^5).

• Средний (или ожидаемый) случай: O(N^3).

В реальности большое влияние оказывают:

1. Быстрые проверки для чётных N (N%2 == 0) и кратных 3 (N%3 == 0), когда решение сразу возвращается (4 или 6 квадратов), фактически за O(1).

2. Предварительная расстановка трёх крупных квадратов (pre), существенно сокращающая свободную зону для дальнейшего перебора.

3. «Отсечения» (count >= best[0]) при достижении уже известного числа квадратов.

Эти эвристики на практике заметно уменьшают глубину перебора, зачастую давая приблизительную оценку O(N^3).

Пространственная сложность

• Сетка: O(N^2).

Программа хранит двумерный список grid размером N×N, где каждая ячейка указывает, занята ли она (1) или свободна (0).

• Рекурсивный стек: до O(N^2).

При неудачном раскладе алгоритм может углубляться практически на каждую клетку (если ставить много маленьких квадратов 1×1). Каждая ветвь рекурсии — это новый уровень в стеке вызовов.

• Список текущих квадратов: до O(N^2).

В списке sol в худшем случае может храниться по одному квадрату на клетку (каждый со стороной 1).

Таким образом, общий объём памяти, который может потребоваться для работы алгоритма, оценивается как O(N^2).

Визуализация

Для визуализации работы алгоритма была использована библиотека matplotlib

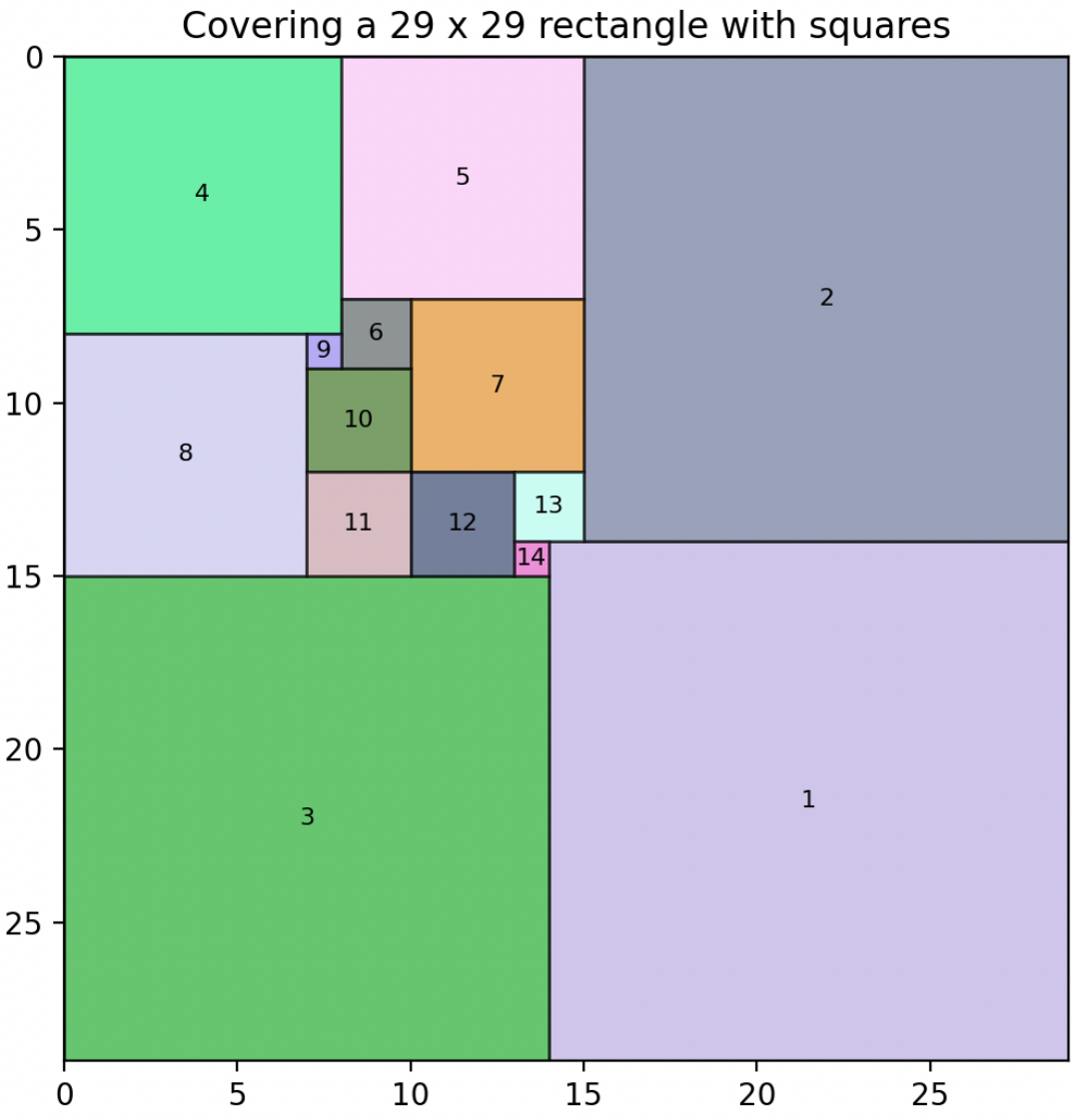


Рис. 1 Визуализация работы алгоритма.

Тестирование

Таблица 1. Тестирование.

Входные данные	*Выходные данные*
7	9 4 4 4 5 1 3 1 5 3 1 1 2 3 1 2 1 3 2 3 3 1 4 3 1 3 4 1
15	6 1 1 10 11 1 5 1 11 5 11 6 5 6 11 5 11 11 5
20	4 1 1 10 11 1 10 1 11 10 11 11 10
37	15 1 1 19 1 20 18 20 1 18 19 20 2 19 22 5 19 27 11 20 19 1 21 19 3 24 19 8 30 27 3 30 30 8

	32 19 6 32 25 1 32 26 1 33 25 5

Исследование

В ходе лабораторной работы было проведено исследование Работоспособности адаптированного алгоритма для прямоугольного поля.

Алгоритм предназначен для нахождения минимального набора квадратов, которые целиком покрывают прямоугольную область размером N×M. При этом допускается использование квадратов разного размера и размещение их в любых позициях, лишь бы они не пересекались. Основой решения служит рекурсивный перебор (backtracking) с отсечениями — после каждой удачно установленной «плитки» алгоритм пытается покрыть оставшиеся свободные клетки, но если текущее число квадратов уже не может улучшить известное решение, перебор данной ветки прекращается.

1. Унификация ориентации прямоугольника

Если M оказывается больше, чем N, то стороны меняются местами: теперь внутренний код всегда работает так, как будто «высота» N \ge «ширина» M. Это делается, чтобы сократить количество дублирующих «горизонтальных» и «вертикальных» случаев. По завершении поиска координаты квадратов, если нужно, «транспонируются» обратно — (x, y)\mapsto(y, x) — чтобы соответствовать изначальному виду.

2. Структуры данных

• Двумерный список grid размером N×M, где каждая ячейка может быть свободна (0) или занята (1).

• Список sol, где временно хранятся все расставленные квадраты в формате (x,y,size), с координатами и длиной стороны.

• Переменная best, содержащая лучшее (минимальное) число квадратов, найденное на данный момент, и соответствующий набор квадратов.

3. Основная логика (backtracking)

• Функция backtrack(count) ищет первую свободную клетку (через find_empty). Если такой клетки нет, значит весь прямоугольник покрыт, и алгоритм обновляет лучшее найденное решение.

• Иначе определяется максимально возможная сторона квадрата max_s (она не должна выходить за границы N×M).

• Перебираются все размеры квадратов от max_s до 1:

1. Проверяется, можно ли поставить такой квадрат (функция can_place).

2. Если разместить можно, «занимаем» клетки (place(..., 1)), добавляем квадрат в sol и рекурсивно вызываем backtrack(count + 1).

3. По возвращении «откатываем» изменения (удаляем квадрат из sol, освобождаем клетки).

• При этом, если count достиг или превысил текущее лучшее значение, ветка перебора сразу прерывается (отсечение).

4. Завершение и итоговый результат

• Когда перебор завершается, все найденные варианты «перебраны», и в best хранится действительно минимальное количество квадратов, а также точный список их координат и размеров.

• Если изначально прямоугольник был «перевёрнут» (M > N), в финале координаты транспонируются обратно, чтобы результат соответствовал входным данным.

5. Визуализация

• После получения результата (списка квадратов) вызывается функция visualize_tiling(N, M, squares).

• Она строит графическое окно, где каждая клетка отображается в системе координат с осями 0..M (по X) и 0..N (по Y), а каждый квадрат рисуется с помощью matplotlib.patches.Rectangle.

• Благодаря ax.invert_yaxis() верхняя левая клетка прямоугольника соответствует точке (0, 0) сверху слева, что удобнее воспринимать, если считать (x, y) от (1,1).

6. Используемые оптимизации

1. Проверка на квадрат: если N = M, вызывается специальная функция solve_square_legacy, которая быстрее разбивает квадрат с помощью отдельных эвристик (половинирование, разбиение на трети).

2. Перестановка N и M, если M > N. Это упрощает логику, так как в остальных проверках предполагается, что N \ge M.

3. Разбиение на полосы, если N кратно M. В этом случае весь прямоугольник можно покрыть вертикальными «столбцами» квадратиков со стороной M без рекурсий.

4. Разбиение на два квадрата, если N = 2 \times M. Тоже тривиальный случай — достаточно двух квадратов.

5. Предварительное размещение крупных квадратов, если (N - M) > (N // 2). Ставятся один-два больших квадрата со стороной M (если позволяют размеры), и только затем запускается рекурсивная доукладка оставшегося пространства.

6. Универсальный рекурсивный backtracking (последний «else»), если все предыдущие быстрые случаи не подходят. Внутри рекурсии есть дополнительное отсечение (пропуск дальнейших поисков), если текущее число квадратов уже не лучше текущего best[0].

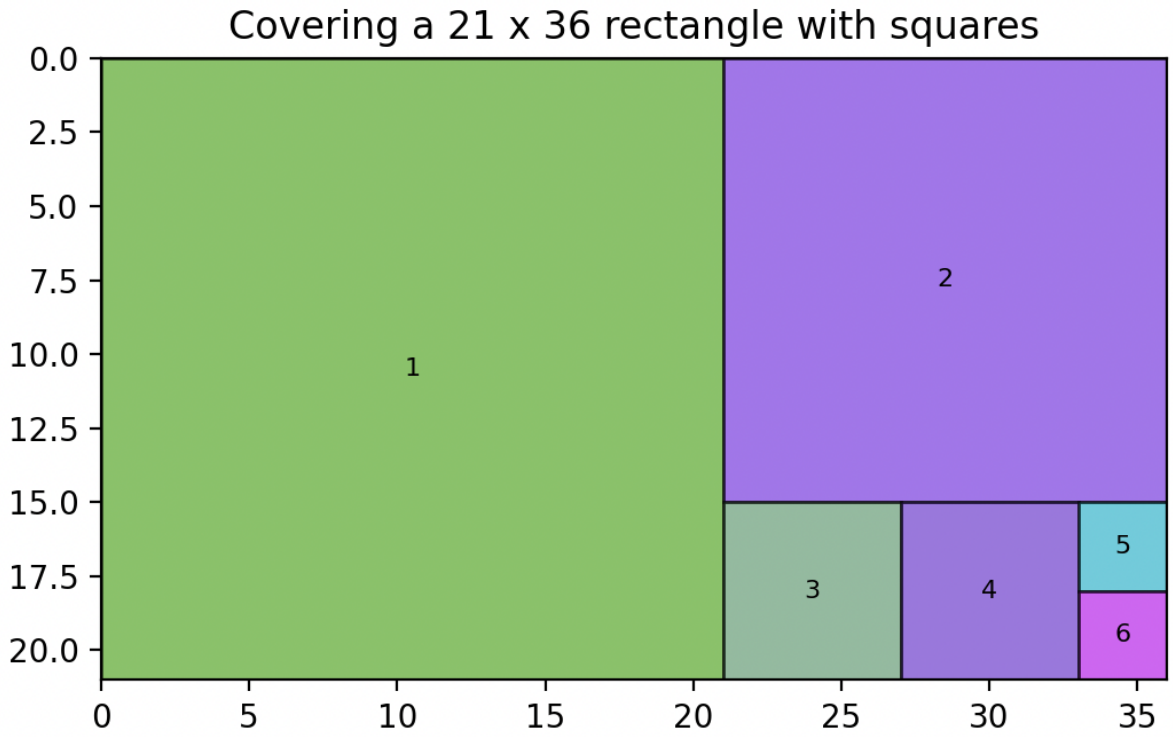
7. Анализ сложности

• В худшем случае, при больших N и M, если значительная часть пространства не «отсекается», перебор может расти экспоненциально, поскольку задача покрытия прямоугольника квадратами относится к NP-трудным.

• В лучшем случае (например, если N = M и используется быстрый алгоритм для квадрата, либо если весь прямоугольник быстро покрывается немногими крупными квадратами), решение получается за считанные итерации.

• В реальной практике многие наборы (N, M) покрываются довольно быстро благодаря отсечениям (когда текущее count уже не может улучшить известное best) и тому факту, что крупные квадраты быстро «закрывают» бóльшую часть прямоугольника.

Таким образом, данная функция реализует универсальный рекурсивный метод покрытия прямоугольника N * M квадратами, не полагаясь на дополнительные «заготовленные» схемы. Несмотря на потенциально высокую теоретическую сложность, практические эвристики (отсечения, большие квадраты в первую очередь) позволяют достичь относительно эффективного перебора, особенно при небольших N и M.



Визуализация работы алгоритма

Вывод

В ходе лабораторной работы была написана программа с использованием метода backtracking. Также было проведено тестирование на различных входных данных. По результатам исследования можно заключить, что число операций растет экспоненциально в зависимости от размера стороны квадрата.