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. 3388	__________________	Еникеев А.А.
Преподаватель	__________________	Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы

Решение задачи поиска минимального разбиения квадрата на конечное число меньших квадратов с целыми сторонами, используя поиск с возвратом: итеративный бэктрекинг. Оптимизация поиска и исследование времени выполнения от размера квадрата.

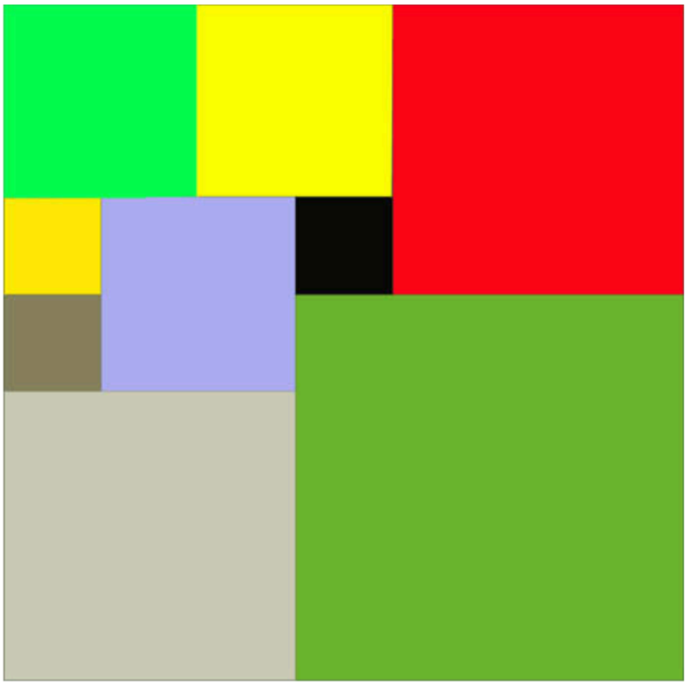
Задание

Вар. 2и

Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

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 ($1 \leq x, y \leq$ N) и длину стороны соответствующего обрезка (квадрата).

Выполнение работы

Структура кода программы

1. Класс *SquareCutter*

Класс, реализующий итеративный бэктрекинг для поиска оптимального разбиения квадрата.

1.1. Поля класса

- *N* (*int*) — размер стороны квадрата.
- *queue* (*deque*) — очередь состояний для перебора.
- *occupied* (*int*) — битовая маска занятых ячеек.
- *best_count* (*int*) — наименьшее найденное число квадратов.
- *best_solution* (*List[Tuple[int, int, int]]*) — оптимальное разбиение.

1.2. Методы класса

- *__init__(self, N)*

Инициализирует объект класса, создаёт очередь и начальные параметры.

- *get_splits(self, min_divider)*

Генерирует предопределённое разбиение для чисел с делителями 2, 3, 5.

- *is_occupied(self, x, y)*

Определяет, занята ли ячейка *(x, y)* с использованием битовой маски.

- *try_place(self, x, y, size)*

Пытается разместить квадрат размера *size* в позиции *(x, y)*, изменяя битовую маску.

- *first_prime_factor(self)*

Определяет наименьший простой делитель *N*. Если *N* — простое, возвращает -1.

- *find_empty(self)*

Находит первую свободную ячейку в квадрате.

- *initial_filling(self)*

Выполняет начальное разбиение квадрата: ставит перые 3 квадрата так, чтобы покрыть левую и нижнюю стороны исходного квадрата, тем самым минимизируя площадь для оставшегося разбиения

- *solve(self)*

Основной метод поиска минимального разбиения:

- Если *N* делится на 2, 3 или 5, использует *get_splits()*.
- Иначе выполняет начальное разбиение через *initial_filling()* и итеративный бэктрекинг через очередь *queue*.

- *add_found_solution(self, pieces_placed, i, j, remains, size)*

Добавляет найденное разбиение в очередь и обновляет *best_solution*, если найдено оптимальное разбиение.

- *get_solution(self)*

Запускает алгоритм *solve()* и возвращает оптимальное разбиение.

2. Основной исполняемый код

- Считывает *N* с ввода.
- Создаёт экземпляр класса *SquareCutter* и решает задачу.
- Выводит минимальное количество квадратов и их координаты.

Исходный код программы см. в прил. А.

Описание алгоритма и способа хранения частичных решений

Если сторона квадрата N – составное число, то минимальное разбиение будет соответствовать разбиению квадрата со стороной, равной меньшему простому множителю N, при условии, что стороны квадратов разбиения будут масштабированы.

Если N – простое число, то сразу построить решение не получится. Алгоритм использует известную структуру для разбиения простых квадратов:

выбираются три меньших квадрата таким образом, чтобы при их размещении они заполняли левую и нижнюю стороны исходного квадрата, тем самым минимизируя оставшуюся площадь для разбиения и сокращая большое количество вариантов.

Перебор вариантов происходит итеративно с помощью очереди. Из очереди извлекается последний вариант (состояние), проверяется, будет ли этот вариант лучше оптимального, если разместить ещё один квадрат. Затем происходит поиск первой свободной клетки и итерация по размерам квадратов. Если квадрат выбранного размера можно разместить, размещаем его и добавляем вариант в конец очереди. Перед добавлением состояния в очередь проверяется, является ли вариант окончательным разбиением. Если это так, то происходит проверка, является ли это разбиение оптимальным; состояние в очередь не добавляется.

Частичные решения хранятся в очереди в виде кортежей, содержащих: число - битовую маску (поле), список размещённых квадратов, где элементы – кортежи вида (*x, y, size*), и свободную площадь (остаток).

Оптимизации алгоритма

1. Предварительное разбиение для чисел с делителями 2, 3, 5 позволяет мгновенно находить решение без перебора в этих случаях.
2. Размещение первых трех квадратов в случае простого N минимизирует оставшуюся площадь, сокращая количество вариантов для перебора.
3. Сначала перебираются квадраты максимального размера, что уменьшает глубину поиска. Причем сторона максимального квадрата выбирается с учетом занятых ячеек и оставшейся площади.
4. Если найдено решение, не лучше уже имеющегося, текущий путь сразу отбрасывается.
5. Проверка возможности размещения квадрата на поле и процесс заполнения ячеек поля происходит одновременно: если какая-то ячейка

уже занята происходит возврат к прежнему состоянию (до вызова метода проверки).

Исследование

Оценим время выполнения алгоритма теоретически. Предположим, что на каждом шаге можно выбрать от 1 до $O(N)$ различных размеров квадрата. Это создает разветвление в дереве поиска, где средняя степень ветвления составляет примерно $O(N)$. Общее число состояний можно оценить как размер дерева перебора, глубина которого имеет порядок $O(N^2)$ (в худшем случае). Таким образом, получаем верхнюю оценку $O(N^{NN})$.

Однако на практике отсечения значительно уменьшают количество возможных путей. Если предположить, что из-за ограничений (занятые клетки, отсечения) средняя степень ветвления ограничена константой c_1, то глубина $O(N^2)$ даёт оценку $O(c_1^{NN})$, что демонстрирует экспоненциальный рост.

Помимо дерева состояний, для каждого состояния выполняются следующие циклы: поиск первой свободной клетки, перебор размеров сторон, попытка разместить квадрат на исходном поле. Сложность этих проверок будет полиномиальной, например, в худшем случае поиск свободной клетки займёт $O(N^2)$. Ограничим рост этой полиномиальной части константой c_2, тогда получаем $O(N^{c2})$.

Получаем $O(c_1^{NN} \cdot N^{c2})$, где c_1, c_2 - некоторые константы. Поскольку c_1^{NN} начинает расти быстрее, начиная с какого-то N, то итоговая сложность будет определяться экспоненциальным ростом.

На рис. 1 представлен рост времени в зависимости от N, который подтверждает теоретическую оценку об экспоненциальном росте.

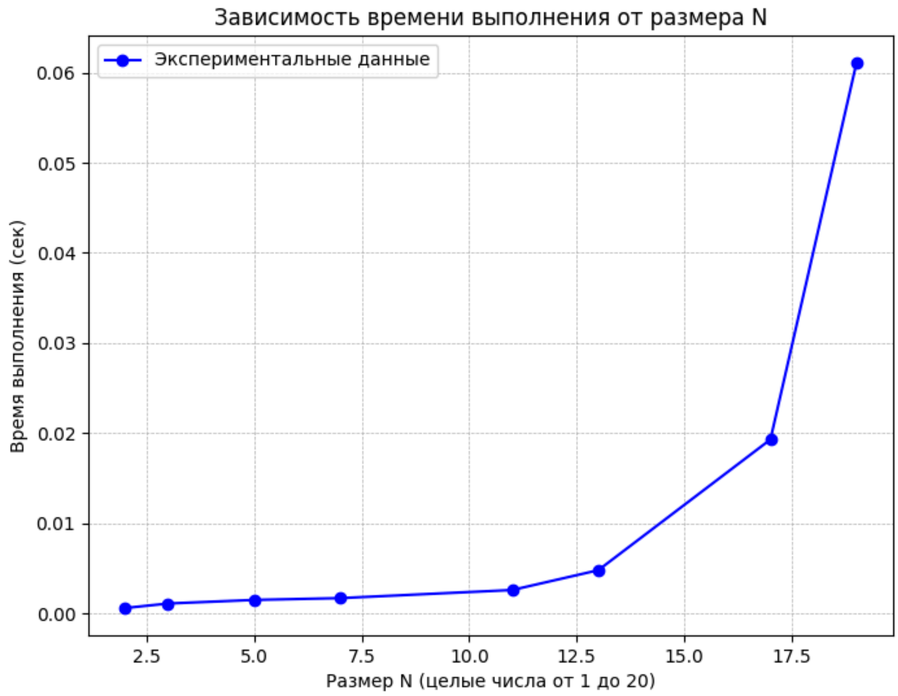


Рис. 1 - Зависимость времени от размера N

Оценим сложность по памяти:

- битовая маска каждого состояния $O(N^2)$, где N - размер стороны квадрата
- пусть K - количество размещенных квадратов для данного состояния, тогда каждый элемент очереди содержит битовую маску $O(N^2)$, список размещенных квадратов $O(K)$ и некоторые константные данные, то есть каждый элемент очереди $O(K + N^2)$
- количество состояний - элементов очереди $O(c_1^{NN})$ (данная оценка рассмотрена выше)
- итого $O(c_1^{NN}(K + N^2))$ - оценка сложности памяти, где N - размер квадрата, K - количество размещенных квадратов, в качество худшего случая $K = N^2$, c_1 - некоторая константа, которая ограничивает степень ветвления

Тестирование

Результаты тестирования программы представлены в табл. 1.

Табл. 1

Входные данные	Выходные данные
19	13 1 1 10 1 11 9 11 1 9 10 11 3 10 14 6 11 10 1 12 10 1 13 10 4 16 14 1 16 15 1 16 16 4 17 10 3 17 13 3
20	4 1 1 10 11 1 10 11 11 10 1 11 10

Выводы

В ходе выполнения лабораторной работы была реализована программа для нахождения минимального разбиения квадрата $N \times N$ на наименьшее число квадратов с целыми сторонами. Алгоритм основан на итеративном поиске с возвратом (бэктрекинге), использующем очередь состояний для перебора возможных разбиений. Анализ алгоритма показал, что в худшем случае его временная сложность имеет экспоненциальный рост. Оценка потребляемой памяти также является экспоненциальной из-за хранения множества состояний поиска.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from collections import deque
from typing import List, Tuple
import time

DEBUG = False

class SquareCutter:
    def __init__(self, N: int):
        """ Инициализация класса для разбиения квадрата """
        self.N = N
        self.queue = deque()  # Стек для хранения состояний
        self.occupied = 0  # Массив занятых ячеек в квадрате
        self.best_count = float('inf')  # Лучшее количество квадратов в решении
        self.best_solution = []  # Лучшее разбиение квадрата.

    def get_splits(self, min_divider):
        """ Возвращает разбиение кв., где сторона - составное число, max = 48 """
        if min_divider == 2:  return [
            (1, 1, N // 2),
            (N // 2 + 1, 1, N // 2),
            (N // 2 + 1, N // 2 + 1, N // 2),
            (1, N // 2 + 1, N // 2)
        ], 4

        if min_divider == 3: return [
            (1, 1, (N * 2) // 3),
            (1, (N * 2) // 3 + 1, N // 3),
            (N // 3 + 1, (N * 2) // 3 + 1, N // 3),
            ((N * 2) // 3 + 1, 1, N // 3),
            ((N * 2) // 3 + 1, N // 3 + 1, N // 3),
            ((N * 2) // 3 + 1, (N * 2) // 3 + 1, N // 3)
        ], 6
        if min_divider == 5: return [
            (1, 1, (N * 3) // 5),
            ((N * 3) // 5 + 1, (N * 2) // 5 + 1, (N * 2) // 5),
            ((N * 3) // 5 + 1, 1, (N * 2) // 5),
            (1, (N * 3) // 5 + 1, (N * 2) // 5),
            ((N * 2) // 5 + 1, (N * 3) // 5 + 1, N // 5),
            ((N * 2) // 5 + 1, (N * 4) // 5 + 1, N // 5),
            ((N * 3) // 5 + 1, (N * 4) // 5 + 1, N // 5),
            ((N * 4) // 5 + 1, (N * 4) // 5 + 1, N // 5)
        ], 8

        return [], -1
```

```
        def is_occupied(self, x: int, y: int) -> bool:
            """ Проверяет, занята ли ячейка (x, y) """
            index = x * self.N + y
            return bool(self.occupied & (1 << index))

        def try_place(self, x: int, y: int, size: int) -> bool:
             """ Пробует разместить квадрат размера size на позиции
(x, y) """
            if x + size > self.N or y + size > self.N:
                return False

            for i in range(x, x + size):
                for j in range(y, y + size):
                    index = i * self.N + j
                    if not bool(self.occupied & (1 << index)):
                        self.occupied |= (1 << index)
                    else:
                        return False
            return True

        def first_prime_factor(self) -> int:
            """ Находит первый простой делитель числа n """

            if self.N % 2 == 0:
                return 2

            for i in range(3, int(self.N**0.5) + 1, 2):
                if self.N % i == 0:
                    return i

            return -1 # число простое

        def find_empty(self) -> Tuple[int, int]:
            """ Ищет первую пустую ячейку и возвращает её координаты
"""
            for i in range(self.N - (self.N + 1) // 2, self.N):
                for j in range(self.N - (self.N + 1) // 2, self.N):
                    if not self.is_occupied(i, j):
                        return i, j
            return -777, -777

        def initial_filling(self) -> None:
            """ Начальное разбиение """

            squares = []

            first_square = (self.N + 1) // 2
            second_square = self.N - first_square # self.N - (self.N
+ 1) // 2

            self.try_place(0, 0, first_square)
            squares.append((1, 1, first_square))

            self.try_place(0, first_square, second_square)
            squares.append((1, first_square + 1, second_square))
```

```
            self.try_place(first_square, 0, second_square)
            squares.append((first_square + 1, 1, second_square))

                self.queue.append((self.occupied, squares, self.N *
self.N - first_square ** 2 - 2 * (second_square ** 2)),)

        def solve(self) -> None:
                """ Решает задачу разбиения квадрата на минимальное
количество меньших квадратов """

            min_divider = self.first_prime_factor()

            if min_divider != -1:
                              self.best_solution, self.best_count =
self.get_splits(min_divider)
                return

            self.initial_filling()

            while self.queue:
                             cur_occupied, pieces_placed, remains =
self.queue.pop()

                if len(pieces_placed) + 1 >= self.best_count:
                    if DEBUG:
                           print(f'remove partition {pieces_placed},
len = {len(pieces_placed)}')
                    continue

                self.occupied = cur_occupied
                i, j = self.find_empty()

                 max_size = min(self.N - max(i, j), self.N - (self.N
+ 1) // 2)

                for size in range(max_size, 0, -1):
                     if size * size <= remains and self.try_place(i,
j, size):
                        self.add_found_solution(pieces_placed, i, j,
remains, size)

                      self.occupied = cur_occupied  # Восстановление
состояния

         def add_found_solution(self, pieces_placed: List[Tuple[int,
int, int]], i: int, j: int, remains: int, size: int) -> bool:
            """ Добавляет новое решение """
            new_pieces = pieces_placed.copy()
            new_pieces.append((i + 1, j + 1, size))
            remains -= size * size

            if remains == 0:
                if DEBUG:
                        print(f'found partition {new_pieces}, len =
{len(new_pieces)}')
                self.best_count = len(new_pieces)
```

```
                self.best_solution = new_pieces
                return True

            if DEBUG:
                        print(f'add partition {new_pieces}, len =
{len(new_pieces)}')

                 self.queue.appendleft((self.occupied, new_pieces,
remains))
            return False

          def get_solution(self) -> Tuple[int, List[Tuple[int, int,
int]]]:
            self.solve()
            return self.best_count, self.best_solution

    def ask_debug_mode():
            answer = input("Использовать режим отладки? [y/n]:
").lower()
         if answer == 'n':
           return False
         else:
           return True


    if __name__ == "__main__":
        N = int(input().strip())
        DEBUG = ask_debug_mode()
        start_time = time.perf_counter()
        solver = SquareCutter(N)
        count, solution = solver.get_solution()
        print(count)
        for square in solution:
            print(*square)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Функция выполнилась за {run_time:.4f} секунд")
```