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: Перебор с возвратом

Студент гр. 3388 _____ Ижболдин А.В.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Цель данной работы заключается в детальном изучении поставленной задачи, а также в разработке эффективного алгоритма, который способен находить её оптимальное решение в пределах заданного времени. Важной частью исследования является анализ зависимости времени выполнения алгоритма от размера квадрата, что позволит оценить его производительность и эффективность при увеличении входных данных.

Задание

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

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 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

Код решает задачу разбиения большого квадрата на меньшие квадраты таким образом, чтобы покрыть всю площадь без наложений и с минимальным количеством квадратов. Основная идея состоит в том, чтобы сначала получить неидеальное (жадное) решение, а затем с помощью перебора с отсечением неоптимальных решений искать лучшее разбиение.

Описание каждой функции с указанием её названия и кратким пояснением, что она делает:

can_place_square - Проверяет, можно ли разместить квадрат со стороной w с координатами верхнего левого угла (x, y) на доске размера n × n. Функция сначала убеждается, что квадрат не выходит за границы, а затем с помощью битовых операций проверяет, что все клетки внутри квадрата ещё свободны.

fill_square - «Заполняет» доску квадратом размера w, начиная с координат (x, y). Функция обновляет битовое представление строк доски, устанавливая соответствующие биты, что означает, что клетки стали занятыми.

find_empty - Ищет первую незаполненную ячейку на доске. Проходя по строкам, функция определяет, где ещё остаются свободные клетки, и возвращает координаты (x, y) первой найденной пустой позиции. Если все клетки заняты, возвращает (-1, -1).

get_greedy_solution - Строит начальное (жадное) решение задачи. Сначала размещаются три квадрата фиксированных размеров, которые разбивают исходный квадрат на части, а затем алгоритм последовательно находит первую свободную ячейку и ставит в неё максимально возможный квадрат. Это решение используется как верхняя граница для оптимизации.

squaring - Выполняет поиск оптимального разбиения квадрата на меньшие квадраты с минимальным их количеством, используя метод branch and bound. Функция начинает с жадного решения, затем перебирает все

варианты (сохраняя частичные решения в стеке), отбрасывая те ветви, где уже размещённых квадратов больше или равно текущему лучшему решению, а также учитывая минимально необходимое количество квадратов для заполнения оставшейся площади.

get_div - Находит наименьший делитель числа n (начиная с 2), если такой имеется, или возвращает n, если число является простым. Это используется для уменьшения (сжатия) задачи: решается задача для меньшего квадрата, а затем решение масштабируется до исходного размера.

solve - Основная функция, которая запускается при выполнении программы. Сначала производится замер времени, затем вызывается get_div для сжатия исходного квадрата. После этого решается задача разбиения с помощью функции squaring, найденное решение масштабируется обратно к исходному размеру, и результаты (количество квадратов, их координаты и время выполнения) выводятся на экран.

Использованные оптимизации

Битовые маски для представления доски. Использование целых чисел для представления каждой строки доски позволяет за одну операцию проверить сразу несколько ячеек. Это существенно ускоряет операции проверки и заполнения квадратов с помощью битовых операций. А также ускоряет процесс копирования доски, так как вместо n^2 операций происходит всего n.

Жадное начальное решение - Применение жадного алгоритма для получения первого приближения дает верхнюю границу по количеству квадратов, что помогает на ранних этапах отбрасывать ветви перебора, которые гарантированно не улучшат результат. Алгоритм пытается заполнить доску максимальными размерами квадратов.

Отсечение ветвей - В процессе перебора с использованием стека производится оценка минимально необходимого числа квадратов для

заполнения оставшейся площади. Если даже при оптимальном заполнении текущая ветка не способна превзойти найденное лучшее решение, она отбрасывается.

Также поиск решения начинается с 3 установленных квадратов (n + 1) // 2 и двух n - (n + 1) // 2.

Сжатие квадрата - Если размер исходного квадрата имеет простой делитель, алгоритм решает задачу для меньшего (сжатого) квадрата (простого делителя), а затем масштабирует найденное решение до исходного размера.

Хранение частичных решений

Частичные решения хранятся в стеке в виде трёхкомпонентных кортежей:

board: список битовых масок, каждая из которых представляет состояние строки (занятые/свободные ячейки). Такой формат позволяет быстро копировать и обновлять состояние доски.

places: список кортежей вида (x, y, w), где x и y – координаты верхнего левого угла размещённого квадрата, а w – его сторона. Этот список описывает уже выполненные ходы.

remain: число, показывающее, сколько ячеек (единичных квадратов) ещё осталось незаполненными.

Исследование времени выполнения от размера квадрата.

Если число не простое, то можно сжать квадрат к минимальному простому делителю, поэтому такие случаи можно быстро просчитать. Однако если N большое простое число, то сложность такой задачи в худшем случае O(2^(n^2)), так как на каждом шаге мы выбираем ставить или нет квадрат 1x1. По памяти сложность будет O(2^(n^2) * (n^2 + k)), то есть максимальное количество расстановок хранимое в один момент умноженное на n^2 памяти для 1 таблица + k расставленных квадратов. Так как оптимизации и

позволяют ускорить алгортим, однако они не влияют на теоритическую сложность, так как рост все равно будет экспоненциальный.

Исследование времени от n равного 2 до 41 (обычный график):

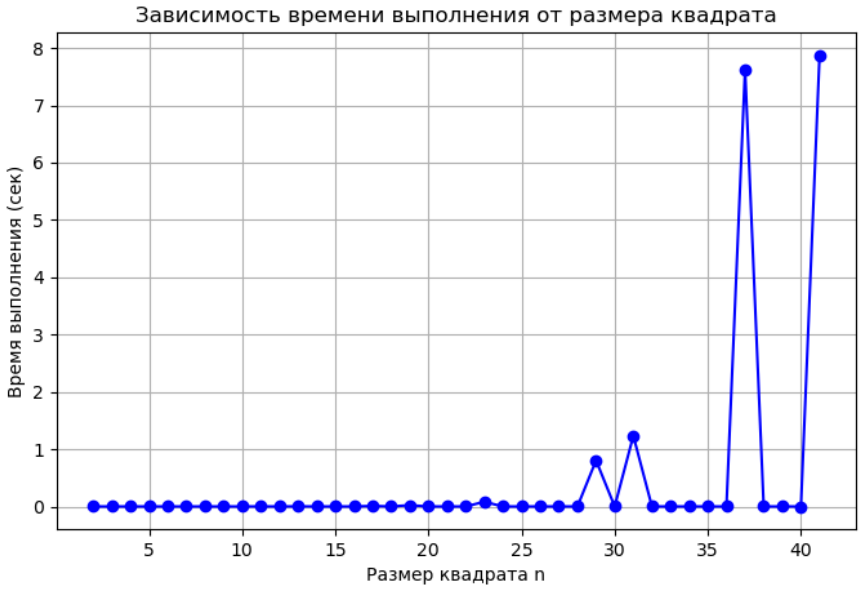

Рис. №1: Исследование времени (обычный график)

Исследование времени от n равного 2 до 41 (логарифм. график):

Рис. №2: Исследование времени (логарифм. график)

Исследование времени от n простых от 2 до 41 (обычный график):

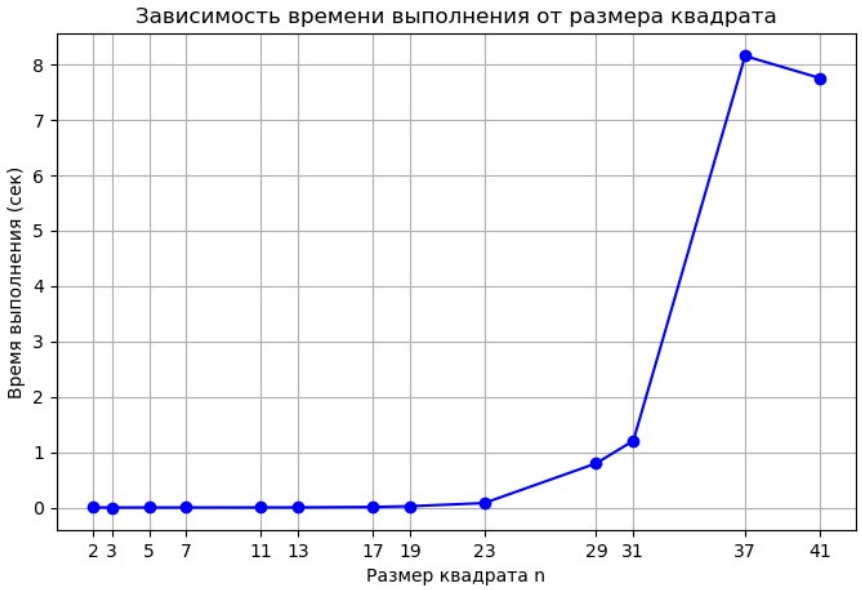

Рис. №3: Исследование времени простых чисел (обычный график)

Исследование времени от n простых от 2 до 41 (логарифм. график):

Рис. №4: Исследование времени простых чисел (логарифм. график)

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: squaring.py

```
    def can_place_square(board: list[int], x: int, y: int, w: int, n:
int) -> bool:
        if x + w > n or y + w > n:
            return False

        mask = ((1 << w) - 1) << (n - x - w)

        for i in range(y, y + w):
            if (board[i] & mask) != 0:
                return False

        return True


    def  fill_square(board:  list[int],  x:  int,  y:  int,  w:  int,  n:
int):
        mask = ((1 << w) - 1) << (n - x - w)

        for i in range(y, y + w):
            board[i] |= mask


    def find_empty(board: list[int], n: int) -> tuple[int, int]:
        for y in range(n):
            row = board[y]
            if row == (1 << n) - 1:
                continue

            masked = (~row) & ((1 << n) - 1)

            if masked == 0:
                continue

            x = n - masked.bit_length()
            return x, y

        return -1, -1


    def get_greedy_solution(n: int, debug = False) -> list[tuple[int,
int, int]]:
        if debug:
            print(f"\n=== Ищем жадное решение ===")
        W = (n + 1) // 2
        board = [0] * n
        places = [(0, 0, W), (0, W, n - W), (W, 0, n - W)]
        remain = n * n

        for x, y, w in places:
            fill_square(board, x, y, w, n)
            remain -= w * w
```

```
            if debug:
                 print(f"Добавляем квадрат: x = {x}, y = {y}, size =
{w}")

        while remain > 0:
            x, y = find_empty(board, n)
            if x == -1:
                if debug:
                     print(f"Пустого места больше не осталось, жадное
решение построено")
                break
            max_size = min(n - x, n - y)
             while max_size > 0 and not can_place_square(board, x, y,
max_size, n):
                if debug:
                    print(f"Добавить квадрат с размером {max_size} не
получилось")
                max_size -= 1
            if max_size == 0:
                break
            fill_square(board, x, y, max_size, n)
            if debug:
                 print(f"Добавляем квадрат: x = {x}, y = {y}, size =
{max_size}")
            places.append((x, y, max_size))
            remain -= max_size * max_size

        return places


    def squaring(n: int, debug=False) -> list[tuple[int, int, int]]:
         best_solution = get_greedy_solution(n, debug)  # ищем жадное
решение

        if len(best_solution) > 2 * n:
            best_solution = [None] * (2 * n)

        W = (n + 1) // 2
        places0 = [(0, 0, W), (0, W, n - W), (W, 0, n - W)]
        board0 = [0] * n
        remain0 = n * n

        for x, y, w in places0:
            fill_square(board0, x, y, w, n)
            remain0 -= w * w

        stack = [(board0, places0, remain0)]

        while stack:
            board, places, remain = stack.pop()
            if debug:
                     print(f"\nПереход к ветке с расположением: ",
*places)

            if len(places) >= len(best_solution):
                if debug:
```

```
                        print(f"Текущее размещение неоптимальней чем
лучшее, отбрасываем: ", *places)
                continue

            if not remain:
                if debug:
                        print(f"--> Пустого места не осталось, нашли
решение еще лучше", *places)
                best_solution = places.copy()
                continue

            x, y = find_empty(board, n)

            max_size = min(n - x, n - y)
               min_squares_needed = (remain + max_size * max_size -
1) // (max_size * max_size)
                        if len(places) + min_squares_needed >=
len(best_solution):
                if debug:
                        print(f"Текущее размещение неоптимальней чем
лучшее, отбрасываем: ", *places)
                continue

            for size in range(max_size, 0, -1):
                if not can_place_square(board, x, y, size, n):
                    if debug:
                        print(f"Добавить квадрат с размером {size} не
получилось")
                    continue

                new_board = board.copy()
                fill_square(new_board, x, y, size, n)

                new_remain = remain - size * size

                new_places = places.copy()
                new_places.append((x, y, size))
                if debug:
                    print(f"Добавляем квадрат: x = {x}, y = {y}, size
= {size}")

                stack.append((new_board, new_places, new_remain))

        return best_solution


    def get_div(n: int) -> int:
        d = 2
        while d * d <= n:
            if n % d == 0:
                return d
            d += 1
        return n


    def solve_squaring(n: int, debug = False):
```

```
        d = get_div(n)  # сжатие квадрата до наим. простого множителя
        if debug:
            print(f"Сжимаем квадрат до n={d}")
        ans = squaring(d, debug)

        # восстановление ответа + исправление координат
        res = n // d
        if debug:
               print(f"Восстанавливаем ответ восстанавливая исходный
квадрат, домножив на {res} всех координаты и ширину квадратов")
         ans = [(x * res + 1, y * res + 1, w * res) for x, y, w in
ans]

        return ans

    if __name__ == '__main__':
        n = int(input())
        ans = solve_squaring(n, True)
        print(len(ans))
        for coords in ans:
            print(*coords)
```