**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: 3p

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Кулач Д.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

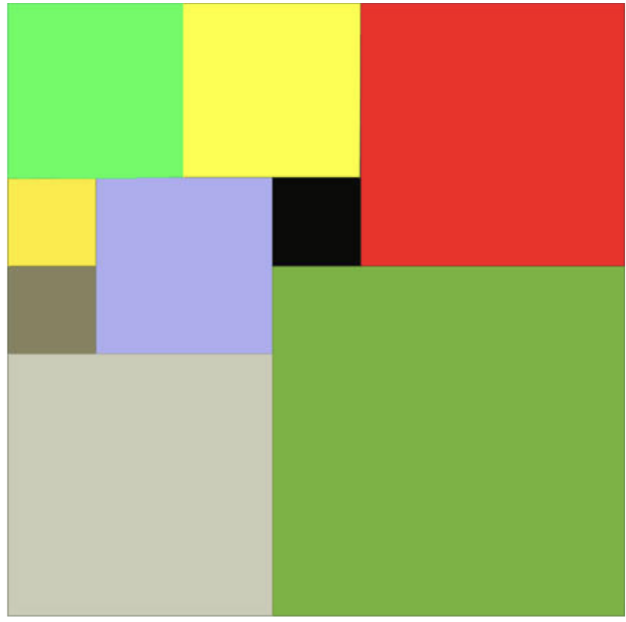
**Цель работы:**

Изучить принцип работы алгоритма поиска с возвратом. Решить с его помощью задачу. Также 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 ≤ N ≤ 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≤*x*,*y*≤*N*) и длину стороны соответствующего обрезка (квадрата).

**﻿Пример входных данных:**7 **Соответствующие выходные данные:**9  
112  
132  
311  
411  
322  
513  
444  
153  
341

**Реализация**

Описание алгоритма:

Для решения поставленной задачи был использован рекурсивный бэктрекинг (рекурсивный поиск с возвратом). Так, после ввода стороны генерируется набор возможных высот для начального размещения крупных квадратов. Этот набор зависит от размера входного значения. Для каждой возможной высоты из набора выполняет следующие действия:

1. Инициализирует начальную диаграмму высот прямоугольника с учётом размещения крупных начальных блоков.
2. Вызывает рекурсивную функцию поиска оптимального решения.
3. Обновляет лучшее решение, если найденное решение лучше текущего лучшего решения.

Описание функций и структур:

* squares\_masks – словарь, в котором для каждого возможного квадрата (x, y, s) (левый верхний угол и размер стороны) хранится битовая маска клеток, которые он покрывает. Это ускоряет проверку наложений.
* find\_first\_free(board, start\_idx) – возвращает индекс первой свободной (непокрытой) клетки в битовой маске board, начиная с позиции start\_idx.
* print\_solution\_matrix(n, result) – формирует и печатает матрицу с номерами квадратов, покрывающих соответствующие ячейки. Используется для отладки и визуализации результата.
* search(board, current\_count, start\_idx, free\_count) – основная рекурсивная функция поиска:
* board – текущая битовая маска занятых клеток;
* current\_count – текущее количество размещённых квадратов;
* start\_idx – индекс для ускоренного поиска свободных клеток;
* free\_count – число оставшихся незанятых клеток.
* best\_count и best\_solution – глобальные переменные, хранящие текущее лучшее (наименьшее) решение.
* dp – словарь, реализующий динамическое программирование: хранит минимальное число квадратов, использованное для каждой маски board, чтобы избежать повторной обработки одних и тех же конфигураций.
* solution – стек текущего решения. После завершения рекурсивного вызова последний квадрат из стека удаляется (обратный шаг бэктрекинга).

Способ хранения частичных решений:

* solution – список, хранящий кортежи (x, y, s) для всех квадратов текущей конфигурации;
* best\_solution[0] – список, содержащий квадраты лучшего (оптимального) покрытия;
* Маска board – битовое представление занятой/свободной области размером N×N.

Алгоритмы оптимизации:

* Битовые маски – обеспечивают быстрые операции проверки пересечений и обновления состояния области;
* Оценка нижней границы (lower\_bound) – минимальное количество квадратов, необходимое для покрытия оставшейся области. Если текущее решение + оценка ≥ лучшее найденное – ветвь обрезается;
* Мемоизация (dp) – исключает повторный перебор одинаковых конфигураций;
* Ограничение максимального размера квадрата – максимальный размер ограничивается (N-1), чтобы избежать лишних проверок;
* Ранний выход – если уже текущий путь не может дать более оптимального решения, выполнение прерывается.

Оценка сложности алгоритма:

Основной идеей алгоритма является рекурсия, соответственно количество возможных переборов будет расти, как степенная функция. Ввиду использования оптимизаций, сложность алгоритма уменьшается на некоторую константу, но в худших всё ещё приближается к экспоненциальной сложности (O(e^n)), где n-сторона квадрата.

**Тестирование**

Таблица 1. Тестирование.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 2 | 4  1 1 1  2 2 1  1 2 1  2 1 1 |
| 3 | 6  1 2 1  2 1 1  1 1 1  2 2 2  1 3 1  3 1 1 |
| 11 | 11  5 6 1  4 6 1  4 3 3  1 4 3  5 1 2  4 2 1  4 1 1  1 1 3  6 6 6  1 7 5  7 1 5 |
| 15 | 6  1 6 5  6 1 5  1 1 5  6 6 10  1 11 5  11 1 5 |
| 19 | 13  9 10 1  8 10 1  8 7 3  4 7 4  1 8 3  5 1 6  4 6 1  4 5 1  1 5 3  1 1 4  10 10 10  1 11 9  11 1 9 |

**Иследование**

Также в ходе лабораторной работы было проведено исследование зависимости количества итераций от стороны квадрата. В ходе исследования получились следующие результаты(рис. 1 и табл. 2).

Таблица 2. Зависимость количества итераций от стороны квадрата.

|  |  |
| --- | --- |
| Сторона квадрата | Количество итераций |
| 3 | 11 |
| 4 | 21 |
| 5 | 71 |
| 6 | 48 |
| 7 | 308 |
| 9 | 212 |
| 11 | 5361 |
| 12 | 181 |
| 13 | 14890 |
| 15 | 971 |
| 17 | 2754 |

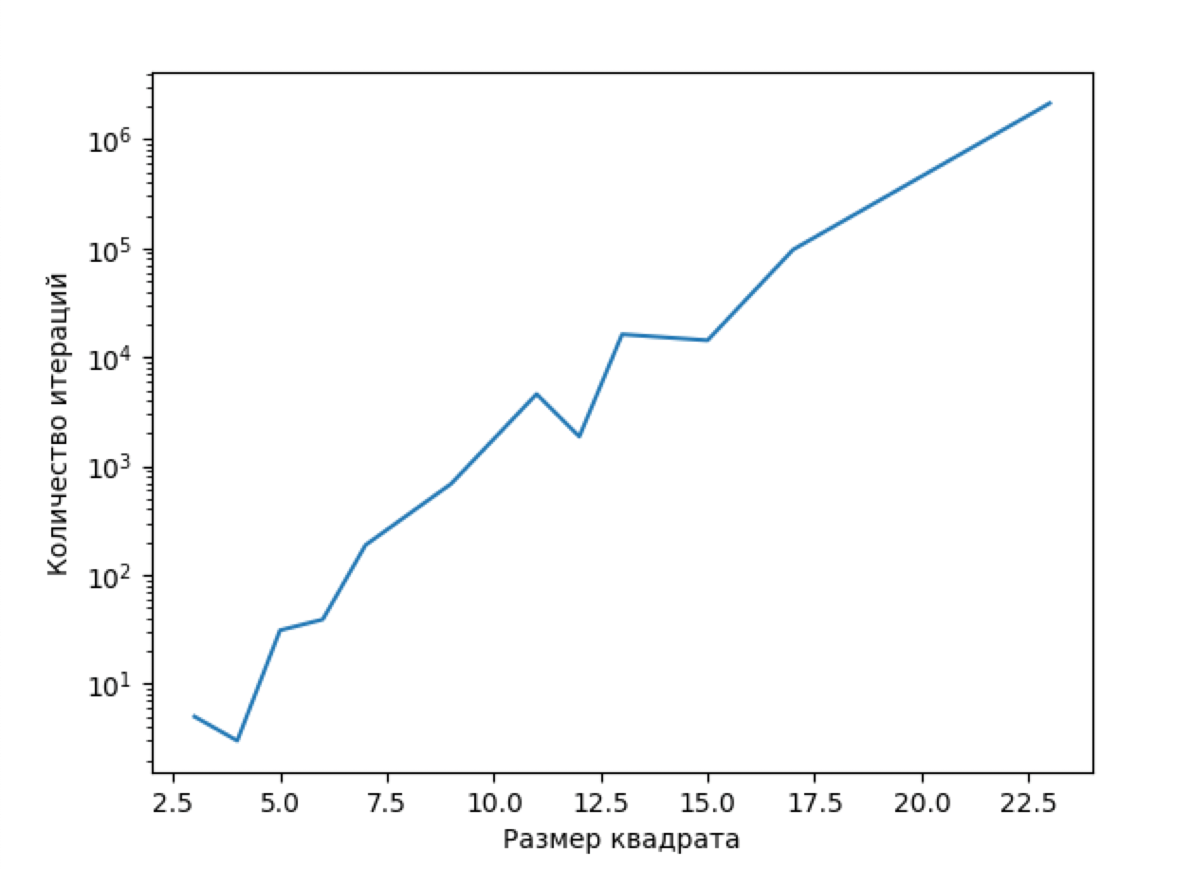


Рис. 1. Зависимость количества итераций от стороны квадрата

Построим логарифмический график зависимости количества итераций от стороны квадрата. Не сложно заметить, что значения в простых числах образуют прямую, что свидетельствует о экспоненциальной зависимости.

**Вывод**

В ходе лабораторной работы была написана программа с использованием алгоритма бэктрекинга. Также было проведено тестирование на различных входных данных. По результатом исследования можно заключить, что зависимость числа операций от размера поля экспоненциальна.

Исходный код программы см. в ПРИЛОЖЕНИИ А.

**ПРИЛОЖЕНИЕ А.**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

main.py

def find\_min\_squares\_with\_debug(N, DEBUG=True):  
 import sys  
 sys.setrecursionlimit(10000)  
  
 total\_cells = N \* N  
 full\_mask = (1 << total\_cells) - 1  
  
 squares\_masks = {}  
 for y in range(N):  
 for x in range(N):  
 max\_side = min(N - x, N - y)  
 for s in range(1, max\_side + 1):  
 if s == N:  
 continue  
 mask = 0  
 for r in range(y, y + s):  
 for c in range(x, x + s):  
 mask |= (1 << (r \* N + c))  
 squares\_masks[(x, y, s)] = mask  
  
 def find\_first\_free(board, start\_idx):  
 for i in range(start\_idx, total\_cells):  
 if ((board >> i) & 1) == 0:  
 return i  
 return -1  
  
 best\_count = [float('inf')]  
 best\_solution = [[]]  
 dp = {}  
 solution = []  
 iteration\_counter = [0]  
  
 def print\_solution\_matrix(n, result):  
 print("\nИтоговая матрица решения")  
 matrix = [[0] \* n for \_ in range(n)]  
 num = 1  
 for (x, y, s) in result:  
 for i in range(s):  
 for j in range(s):  
 matrix[y - 1 + i][x - 1 + j] = num  
 num += 1  
  
 for row in matrix:  
 print(" ".join(f"{cell:2}" for cell in row))  
  
 def search(board, current\_count, start\_idx, free\_count):  
 iteration\_counter[0] += 1  
 iter\_num = iteration\_counter[0]  
  
 if DEBUG:  
 print(f"\nИтерация #{iter\_num}:")  
 print(f"Текущий стек ({len(solution)} квадратов):")  
 for sq in solution:  
 print(f"\tКвадрат: ({sq[0]}, {sq[1]}) размер {sq[2]}")  
  
 if current\_count >= best\_count[0]:  
 if DEBUG:  
 print("\tОбрезка ветви: текущих квадратов уже больше оптимума")  
 return  
 if board == full\_mask:  
 if current\_count < best\_count[0]:  
 best\_count[0] = current\_count  
 best\_solution[0] = solution.copy()  
 if DEBUG:  
 print(f"\tНайден новый лучший результат: {current\_count} квадратов")  
 return  
 if board in dp and dp[board] <= current\_count:  
 return  
 dp[board] = current\_count  
  
 idx = find\_first\_free(board, start\_idx)  
 if idx == -1:  
 return  
 x = idx % N  
 y = idx // N  
  
 max\_possible = min(N - x, N - y)  
 if max\_possible == N:  
 max\_possible = N - 1  
 if max\_possible <= 0:  
 max\_possible = 1  
 max\_area = max\_possible \* max\_possible  
 lower\_bound = (free\_count + max\_area - 1) // max\_area  
 if current\_count + lower\_bound >= best\_count[0]:  
 if DEBUG:  
 print("\tОбрезка ветви: оценка нижней границы >= текущего оптимума")  
 return  
  
 max\_side = min(N - x, N - y)  
 if max\_side == N:  
 max\_side = N - 1  
  
 for s in range(max\_side, 0, -1):  
 mask = squares\_masks.get((x, y, s))  
 if mask is None:  
 continue  
 if board & mask == 0:  
 solution.append((x + 1, y + 1, s))  
 if DEBUG:  
 print(f"\tДобавляем квадрат: ({x + 1}, {y + 1}) размер {s}")  
 search(board | mask, current\_count + 1, idx + 1, free\_count - s \* s)  
 if DEBUG:  
 print(f"\tУбираем квадрат: ({x + 1}, {y + 1}) размер {s}")  
 solution.pop()  
  
 search(0, 0, 0, total\_cells)  
  
 return best\_count[0], best\_solution[0], iteration\_counter[0], print\_solution\_matrix  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 N = int(input("Размер столешницы N: "))  
 if 2 <= N < 20:  
 count, squares, iterations, print\_matrix\_fn = find\_min\_squares\_with\_debug(N, DEBUG=True)  
 print(f"\nМинимальное количество квадратов: {count}")  
 for x, y, s in squares:  
 print(f"{x} {y} {s}")  
 print(f"\nКоличество итераций: {iterations}")  
 print\_matrix\_fn(N, squares)  
 else:  
 print("Недопустимый размер. Введите число от 2 до 19.")