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. 0304	Мажуга Д.Р.
Преподаватель	Фирсов М.А.

Санкт-Петербург,
2022

Цель работы.

Изучить алгоритмы поиска с возвратом на примере задачи разбиения квадрата на минимальное множество меньших квадратов. Изучить программную реализацию итеративного алгоритма бэктрекинга.

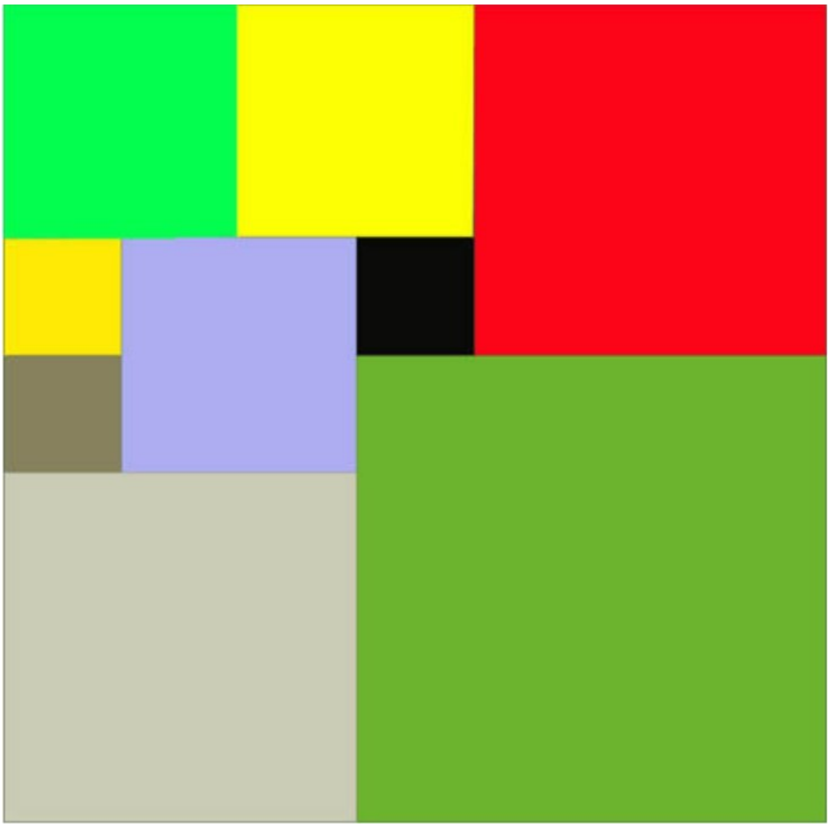
Задание.

Вариант 4и.

Итеративный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

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

311

411

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Описание алгоритма.

Задача решена с помощью итеративной реализации метода бэктрекинга. Найденные частичные решения добавляются в очередь, первое решение из очереди расширяется всеми возможными способами, и каждое новое частичное решение снова добавляется в очередь.

Как только будет найдено решение, поиск останавливается, так как найденное решение гарантированно является одним из минимальных: все решения меньшего размера уже были перебраны.

Поиск возможных расширений текущего решения сводится к поиску размера максимального квадрата, который можно вставить левым верхним углом в данную точку. Текущее решение расширяется всеми квадратами размера от максимального до 1, каждое новое частичное решение отправляется в очередь.

Если вставить еще один квадрат нельзя, значит, свободные места кончились, а решение найдено.

Оптимизации.

1) Квадрат хранится не как двумерный список логических значений, а как одномерный список целых чисел, обозначающих количество свободных клеток в данном столбце. Это ускоряет обработку и снижает асимптотические затраты по памяти.
2) Вставка всегда осуществляется в левую клетку самой верхней из доступных строк. Это многократно снижает количество обрабатываемых частичных решений.
3) Если n — простое, то минимальное решение будет содержать квадраты с длинами сторон $n\ div\ 2 + 1$; $n\ div\ 2$; $n\ div\ 2$. В таком случае очередь решений можно инициализировать частичным решением из трех элементов, а не пустым решением.

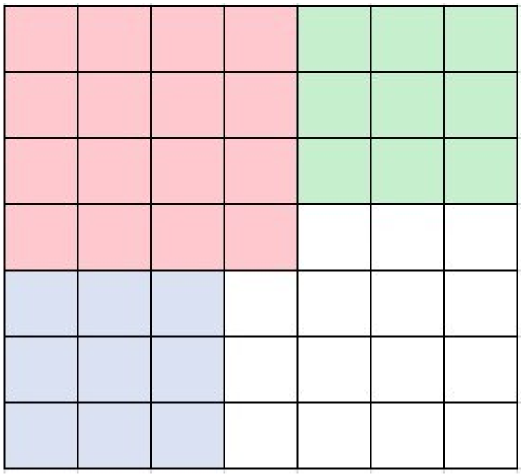


Рисунок 1 — частичное решение для квадрата со стороной 7. Три квадрата фиксированы.

Оценка сложности алгоритма.

Оценим сложность алгоритма по кол-ву операций и используемой памяти без учёта логирования.

Экспериментально выяснено, что длина решения не превышает N + 3 для используемых входных данных.

Сначала состояние квадрата восстанавливается по частному решению: цикл по всем элементам частного решения (менее N + 3) с циклом по размеру вставленного квадрата (меньше N) — $O(N^2)$.

Затем поиск места для вставки нового решения — поиск максимального элемента в списке столбцов и его индекса — O(N). Поиск максимального размера для вставки занимает O(N) времени, а вставка проводится для каждого размера от максимального до 1, не превышающего N – 1. Сложность этого этапа — $O(N^2)$.

Количество частичных решений можно оценить из следующих соображений: для каждого решения будет добавлено не более N – 1 решений, каждое из которых — тоже решение. Так, учитывая ограничение на максимальное число элементов в решении, равное N + 3, $(N - 1)^{N + 3} \rightarrow O(N^N)$. Количество операций, затрачиваемых на обработку частного решения, полиномиально, следовательно, затраты по времени также оцениваются как $O(N^N)$.

Таким образом, алгоритм требует $O(N^N)$ памяти и операций.

Функции и структуры данных. main.py

Для нахождения минимального разбиения прямоугольника используется функция *findMinPartition(width, height, printResult, countSolutionsNum, printDebug)*.

width, *height* — размеры прямоугольника, целые числа. *printResult*, *countSolutionsNum*, *printDebug* — настройки вывода размера в консоль, логический тип.

Данная функция обеспечивает взаимодействие с очередью решений *partialSolutions*, которая представлена ввиде вложенного списка, и вызов остальных вспомогательных функций.

Функция *findInsertion(heights, width, height, printDebug)* возвращает столбец, в который можно вставить квадрат, и размер вставляемого квадрата.

heights — список высот, определяющий состояние прямоугольника.

width, height — размеры прямоугольника, целые числа.

Максимальный размер вставляемого квадрата определяется с помощью функции *findMaxInsertableSize(heights, col, width, height)*. *col* — индекс столбца, в который вставляется квадрат. *width, height* — размеры прямоугольника, целые числа. *heights* — список высот, определяющий состояние прямоугольника.

Функция *heightsFromSolution(curSolution, width, height)* позволяет восстановить состояние квадрата (списка высот) по частичному решению, поочередно выполняя записанные в нем шаги. Возвращает список высот столбцов *heights*.

curSolution — частичное решение, список из кортежей по два элементов, определяющих столбец вставки и размер квадрата.

Функция *isPrime(num)* нужна для оптимизации решения в случае, когда исследуется квадрат со стороной, длина которой — простое число. Принимает целое число, возвращает логическое значение.

Функция *printSolution(solution, width, height)* выводит решение задачи, поэлементно выполняя шаги, записанные в списке кортежей *solution*.

width, height — размеры прямоугольника, целые числа.

test.py

Функция *testSquares()* выводит в консоль время, затраченное на выполнение функции *findMinPartition* для квадратов размера от 2 до 20.

Функция *testRectangles()* выводит в консоль время выполнения функция для набора пар высоты и ширины прямоугольников, а также количество минимальных решений для них.

Индивидуализация.

Перед выполнением алгоритма мы выполняем проверку, явлется ли поданные нам на вход стороны равными, так как для квадрирования квадрата мы использовали следующую оптимизацию: если n — простое, то минимальное решение будет содержать квадраты с длинами сторон $n\ div\ 2 + 1$; $n\ div\ 2$; $n\ div\ 2$, Так как данная оптимизация верна только для квадрата с простой стороной, то прямоугольник находится с помощью итеративной реализации метода бэктрекинга, описанного выше, не выполняя данную оптимизацию.

Тестирование.

	Входные данные	Выходные данные	Комментарий
	16	4 1 1 8 9 1 8 1 9 8 9 9 8	

	7	9 1 1 4 5 1 3 1 5 3 5 4 1 6 4 2 4 5 1 5 5 1 4 6 2 6 6 2	
	19	13 1 1 10 11 1 9 1 11 9 11 10 1 12 10 1 13 10 2 15 10 5 10 11 2 12 11 1 12 12 3 10 13 2 10 15 5 15 15 5	

	2	4 1 1 1 2 1 1 1 2 1 2 2 1	
	12 7	Минимальных решений: 2 Пример решения: 7 1 1 3 4 1 3 7 1 3 10 1 3 1 4 4 5 4 4 9 4 4	Для прямоугольников выводится одно решение и их общее количество.

	3 6	Минимальных решений: 8 Пример решения: 9 1 1 1 2 1 2 1 2 1 1 3 1 2 3 2 1 4 1 1 5 1 2 5 2 1 6 1	

Выводы.

В ходе работы был рассмотрен итеративный вариант метода бэктрекинга для разбиения квадрата на квадраты меньшего размера. Оценка сложности данного алгоритма составляет $O(N^N)$ и по памяти, и по времени.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

1) Файл *main.py*

```
def findMaxInsertableSize(heights, col, width, height):
    curHeight = heights[col]
maxInsertable = 1
    for i in range(1, min(curHeight, len(heights) - col, width -
1, height - 1)):
        # максимальный размер не превышает количество свободных
клеток в столбце и количества столбцов справа        if heights[col +
i] < curHeight:  # если очередной размер
не вмещается            return
maxInsertable
maxInsertable += 1     return
maxInsertable
  def findInsertion(heights, width, height,
printDebug):
    insertInd = heights.index(max(heights))
    # Вернется первое вхождение - левая клетка самой верхней
свободной строки     if heights[insertInd] == 0:
        return 0, 0
    maxSize = findMaxInsertableSize(heights, insertInd, width,
height)     if printDebug:         print(f'Вставка в столбец
{insertInd} квадратов до размера
{maxSize}')     return insertInd,
maxSize
  def heightsFromSolution(curSolution, width,
height):     heights = [height for _ in
range(width)]     for ind, size in curSolution:
for col in range(ind, ind+size):            #
занимаем свободные клетки            heights[col] -
= size     return heights
  def printSolution(solution, width,
height):     heights = [height for _ in
range(width)]     print(len(solution))
for ind, size in solution:
        print(ind + 1, height - heights[ind] + 1, size)
for col in range(ind, ind + size):
            heights[col] -= size
  def
isPrime(num):
if num == 2:
        return False     for divisor in
range(2, int(num**0.5) + 1):
        if num % divisor == 0:
            return False
    return True


def   findMinPartition(width,    height,    printResult,
countSolutionsNum, printDebug):
    solutions = []
```

```
        if width == height and isPrime(width):  # частная оптимизация
для квадрата с простой стороной        partialSolutions = [[(0, width
     // 2 + 1), (width // 2 + 1,
width // 2), (0, width // 2)]]
     else:         partialSolutions
     = [[]]
         while
     partialSolutions:
             curSolution = partialSolutions.pop(0)  # берем элемент из
очереди         if printDebug:
     print(f'Обработка решения {curSolution}')
             heights = heightsFromSolution(curSolution, width, height)
# восстанавливаем состояние из частичного решения         ind,
     maxInsertableSquare = findInsertion(heights, width,
height, printDebug)         if maxInsertableSquare:             for
     insertSize in range(1, maxInsertableSquare + 1):
                     partialSolutions.append(curSolution + [(ind,
insertSize)])  # расширяем частичное решение         else:  # нечего
     вставить только тогда, когда решение
найдено
     solutions.append(curSolution)
                 if not countSolutionsNum:  # если достаточно одного
решения                 if
     printResult:
                         printSolution(curSolution, width, height)
     break
                 elif len(solutions[0]) < len(curSolution):  # если
найдено более длинное решение                 print(f'Минимальных
     решений: {len(solutions) -
1}')                 if printResult:
                         print('Пример решения:')
                         printSolution(solutions[0], width, height)
     break
                 elif len(partialSolutions) == 0:  # если решение -
самое длинное из возможных                 print(f'Минимальных
     решений: {len(solutions)}')                 if printResult:
                         print('Пример решения:')
                         printSolution(solutions[0], width, height)
     break
       if __name__ == '__main__':
     printDebug = False
     countSolutionsNum = False
     printResult = True
         sizes = list(map(int, input().split()))
     if len(sizes) == 1:
             width = height = sizes[0]
     else:
             width, height = sizes

         findMinPartition(width, height, printResult,
countSolutionsNum, printDebug)
```

2) Файл *test.py*

```
     from time import perf_counter_ns from
     main import findMinPartition rom time
```

```
import perf_counter_ns from main
import findMinPartition
  def
testSqares():
    printResult = False
printDebug = False
countSolutionsNum = False     for
i in range(2, 21):
        start = perf_counter_ns()
        findMinPartition(i, i, printResult, countSolutionsNum,
printDebug)         print(f'Размер    квадрата:  {i},  время
           работы:
{(perf_counter_ns() - start) / 1000000} ms.')
  def
testRects():
    printResult = False     printDebug = False
countSolutionsNum = True     for width, height in [(12, 7),
(3, 6), (5, 8), (10, 12)]:
        start = perf_counter_ns()
        print(f'Размер прямоугольника: {width}x{height}')
        findMinPartition(width,       height,       printResult,
countSolutionsNum, printDebug)         print(f'Время работы:
{(perf_counter_ns() - start) / 1000000} ms.')

testSqares()
print('\n\n')

testRects()
  def
testSqares():
    printResult = False
printDebug = False
countSolutionsNum = False     for
i in range(2, 21):
        start = perf_counter_ns()
        findMinPartition(i, i, printResult, countSolutionsNum,
printDebug)         print(f'Размер    квадрата:  {i},  время
работы:
{(perf_counter_ns() - start) / 1000000} ms.')
  def
testRects():
    printResult = False     printDebug = False
countSolutionsNum = True     for width, height in [(12, 7),
(3, 6), (5, 8), (10, 12)]:
        start = perf_counter_ns()
        print(f'Размер прямоугольника: {width}x{height}')
        findMinPartition(width,       height,       printResult,
countSolutionsNum, printDebug)         print(f'Время работы:
{(perf_counter_ns() - start) / 1000000} ms.')

testSqares()
print('\n\n')

testRects()
```