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. 1304 _____ Андреев В.В

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Решить задачу методом поиска с возвратом. Применить оптимизацию методом ветвей и границ на практике.

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 её, собрав из уже имеющихся обрезков (квадратов).

Например, столешница 7 на 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, целое число 2 <= N <= 20.

Выходные данные: K – минимальное число обрезков, из которых можно построить квадрат, и K строк с числами x, y, w, где x, y – координаты обрезка, w – длина обрезка.

Выполнение работы.

Для решения задачи применен метод рекурсивного перебора с возвратом. Его основное отличие от простого перебора заключается в отсечении заведомо неверных решений по принципу ветвей и границ с возвратом к предыдущему состоянию.

Используемые оптимизации:

1. Для размера поля кратного 2 и 3 ответ заведомо известен. Так для кратных 2 это 4 квадрата с длинной стороны N / 2.
2. Для простых чисел известно оптимальное начало решения, где в одном углу стоит квадрат размером (N + 1) / 2, в 2-ух соседних прилежащих

углах стоят квадраты размером (N + 1) / 2 — 1. Что позволяет сократить перебор до четверти исходного квадрата.

3. Квадраты хранятся не матрицей, а координатами с размерами, что в худшем случае будет эквивалентно полному заполнению поля матрицы. Таким образом добавление квадратов происходит со сложностью добавления элемента в конец массива, что также позволяет легко убирать последний добавленный квадрат.
4. Поле хранит оставшуюся незанятую площадь, по значению которой можно отбрасывать случаи перебора, которые превышают доступную площадь.
5. Отбрасываются случаи, где текущее количество квадратов >= найденного результата.
6. Квадраты размера 1 не порождают новых ветвей в переборе. Также количество квадратов со стороной 1 легко сравнивать с оставшейся свободной площадью. Так можно отбрасывать случаи где текущее количество квадратов + свободная площадь >= лучший найденный случай.
7. Рекурсия оптимизируется отсутствием копирования поля между вызовами. Вызывающая сторона заботится о добавлении квадрата в поле и удаления его после отработки ветки перебора. Это легко достигается за счет стекообразного хранения квадратов в поле.
8. По империческим оценкам было выяснено, что количество квадратов для стороны столешницы < 20 не превышает 13. Этот факт использовался для отсечения вариантов, в которых > 13 квадратов.

Используемые классы и функции:

Класс *Square* – хранит информацию о квадрате (координаты и размер)

Метод *Square::IsCollide* - проверяет пересечение квадратов.

Класс *SquareMap* – поле квадратов. Хранит массив квадратов и свободную площадь.

Метод *SquareMap::CanAddSquare* – проверяет, что возможно добавить квадрат. Проверяет на пересечение со всеми добаленными квадратами.

Метод *SquareMap::AddSquare* – добавляет квадрат без проверки на корректность.

Метод *SquareMap::Pop* – удаляет последний добавленный квадрат.

Метод *SquareMap::IsFilled* – проверяет на заполненность поля (свободная площадь == 0*)*

Функция *TryToSetResult* - устанавливает в глобальную переменную переданное поле в случае если в нем меньше квадратов.

Функция FindPositions – рекурсивный поиск с возвратом. Результат записывается и сравнивается со значением в глобальной переменной GsearchResult.

В качестве хранения ответа и промежуточных данных используется глобальная переменная поля *GSearchResult*. Исходный код приведен в приложении А.

Выводы.

Написан алгоритм, находящий минимальное разбиение квадрата с заданной стороной. Изучен и применен метод ветвей и границ. Успешно пройдены тесты на платформе Stepik для N <= 20. Применен ряд оптимизаций для успешного выполнения задания за указанное время, а именно: генерация готового разбиения для N кратного 2 и 3; вставка трех квадратов заданного размера, вычисляемого по N, для простого N; ограничение максимально возможного количества квадратов в разбиении; оптимальное хранение квадратов в поле.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from copy import deepcopy


'''
	Map primitive. Provides simple functional for square manipulating.
'''
class Square:
	def __init__(self, X: int, Y: int, R: int):
		self.X = X
		self.Y = Y
		self.R = R

	'''
		Check that this square is collide with other.

		@param S - other square.
		@return true if collide.
	'''
	def IsCollide(self, S) -> bool:
		return (S.X < self.X + self.R) and (S.Y < self.Y + self.R) and
(S.X + S.R > self.X) and (S.Y + S.R > self.Y)
	'''
		@return square size.
	'''
	def GetSize(self) -> int:
		return self.R * self.R

	'''
		Print square data to stdout.
	'''
	def Print(self) -> None:
		print(self.X, self.Y, self.R)

'''
	Container of used squares.
'''
class SquareMap:
	def __init__(self, N: int):
		self.N = N
		self.__FreeSize = N * N
		self.__SquaresList = []
		self.__OnesSquareList = []

	'''
		Check that can add given square.

		@param S - square to check.
		@return true if given square can be added.
	'''
	def CanAddSquare(self, S: Square) -> bool:
		if (self.__FreeSize - S.GetSize() < 0) or (S.X + S.R > self.N
+ 1) or (S.Y + S.R > self.N + 1):
			return False
```

```
            for LS in self.__SquaresList:
                  if LS.IsCollide(S):
                        return False
            for LS in self.__OnesSquareList:
                  if LS.IsCollide(S):
                        return False
            return True
      '''
            Add given square without checking.
      '''
      def AddSquare(self, S: Square) -> None:
            self.__FreeSize -= S.GetSize()
            if S.R == 1:
                  self.__OnesSquareList.append(S)
            else:
                  self.__SquaresList.append(S)

      '''
            Remove last added square.
      '''
      def Pop(self) -> None:
            self.__FreeSize += self.__SquaresList[-1].GetSize()
            self.__SquaresList.pop()
      '''
            Remove all squares with size 1.
      '''
      def ClearOnes(self) -> None:
            self.__FreeSize += len(self.__OnesSquareList)
            self.__OnesSquareList.clear()

      '''
            Check that map is complete filled.
      '''
      def IsFilled(self) -> bool:
            return self.__FreeSize <= 0
      '''
            @return count of squares in map.
      '''
      def GetCount(self) -> int:
            return len(self.__SquaresList) + len(self.__OnesSquareList)
      '''
            @return unused square size of map.
      '''
      def GetFreeSize(self) -> int:
            return self.__FreeSize

      '''
            Print all containing squares to stdout.
      '''
      def Print(self) -> None:
            for LS in self.__SquaresList:
                  LS.Print()
            for LS in self.__OnesSquareList:
                  LS.Print()


'''
      Result of search will be here.
'''
GSearchResult = None

'''
      Try to set new result of search if it is better than current.
```

```
      '''
      def TryToSetResult(Map: SquareMap) -> None:
            global GSearchResult
            if (GSearchResult is None) or (Map.GetCount() <
GSearchResult.GetCount()):
                  GSearchResult = deepcopy(Map)
      '''
            Main backtracking function.
      '''
      def FindPositions(Map: SquareMap, InitR: int) -> bool:
            global GSearchResult

            if Map.IsFilled():
                  TryToSetResult(Map)
                  return True
            else:
                  if Map.N <= 25:
                        if Map.GetCount() >= 13:
                              return False
                  if (not GSearchResult is None) and (Map.GetCount() + 1 >=
GSearchResult.GetCount()):
                        return False

            for r in range(InitR, 0, -1):
                  if r * r > Map.GetFreeSize():
                        continue
                  if r > 1:
                        LSkipXY = False
                        for x in range(1, Map.N - r + 2):
                              for y in range(1, Map.N - r + 2):
                                    LSquare = Square(x, y, r)
                                    if not Map.CanAddSquare(LSquare):
                                          continue
                                    Map.AddSquare(LSquare)
                                    if not FindPositions(Map, InitR):
                                          Map.Pop()
                                          return True
                                    else:
                                          Map.Pop()
                                          LSkipXY = True
                                          break
                              if LSkipXY:
                                    break
                  else:
                        if Map.N <= 25:
                              if Map.GetFreeSize() + Map.GetCount() > 13:
                                    return False
                        if (not GSearchResult is None) and (Map.GetFreeSize() +
Map.GetCount() >= GSearchResult.GetCount()):
                              return False

                        for x in range(1, Map.N + 1):
                              for y in range(1, Map.N + 1):
                                    LSquare = Square(x, y, 1)
                                    if not Map.CanAddSquare(LSquare):
                                          continue
                                    Map.AddSquare(LSquare)
                        TryToSetResult(Map)
                        Map.ClearOnes()
                        return True
            return False
```

```
if __name__ == "__main__":
      N = int(input())
      GSearchResult = None
      if N % 2 == 0:
            print(4)
            print(1, 1, N // 2)
            print(N // 2 + 1, 1, N // 2)
            print(1, N // 2 + 1, N // 2)
            print(N // 2 + 1, N // 2 + 1, N // 2)
      elif N % 3 == 0:
            print(6)
            print(1, 1, N // 3 * 2)
            print(1, N // 3 * 2 + 1, N // 3)
            print(N // 3 + 1, N // 3 * 2 + 1, N // 3)
            print(N // 3 * 2 + 1, 1, N // 3)
            print(N // 3 * 2 + 1, N // 3 + 1, N // 3)
            print(N // 3 * 2 + 1, N // 3 * 2 + 1, N // 3)
      elif N in [5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
59]:
            LMap = SquareMap(N)
            LMap.AddSquare(Square((N + 1) // 2, (N + 1) // 2, (N + 1) //
2))
            LMap.AddSquare(Square((N + 1) // 2 + 1, 1, (N + 1) // 2 - 1))
            LMap.AddSquare(Square(1, (N + 1) // 2 + 1, (N + 1) // 2 - 1))
            FindPositions(LMap, N // 4 + 1)
            print(GSearchResult.GetCount())
            GSearchResult.Print()
      else:
            FindPositions(SquareMap(N), N - 1)
            print(GSearchResult.GetCount())
            GsearchResult.Print()
```