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 _____ Стародубов М.В.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Реализация алгоритма, выполняющего решение поставленной задачи с применением поиска с возвратом.

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).

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).

Выполнение работы.

Исходный код программы находится в приложении А.

Реализован класс *Desk*, являющийся моделью столешницы, которая в процессе работы алгоритма заполняется обрезками. Класс *Desk* имеет следующие поля: *_size* — размер столешницы, передается в качестве аргумента конструктора класса; *_next_unfilled_position* — пара значений, задающая самую верхнюю левую незаполненную обрезком позицию на столешнице; *_max_piece_size* — максимальный размер обрезка, левый верхний угол которого

находится на позиции *_next_unfilled_position*; *_pieces* — динамический массив, хранящий в себе обрезки, расположенные на столешнице, каждый обрезок задается как тройка значений *x*, *y*, *w*, где *x* и *y* — позиция левого верхнего угла обрезка на столешнице, *w* — резмер стороны обрезка.

Для создания объектов класса *Desk* реализован конструктор, принимающий в качестве аргумента размер стороны столешницы, также был реализован конструктор копирования.

Для добавления обрезка на столешницу реализован метод *add_piece*. В качестве аргументов данный метод принимает размер обрезка и позицию его верхнего левого угла, при передаче методу только размера добавляемого обрезка он будет помещен на позицию, сохраненную в поле *_next_unfilled_position*. Добавление обрезка происходит следующим образом: сначала обрезок записывется в динамический массив *_pieces*, после чего происходит обновление значений полей *_next_unfilled_position* и *_max_piece_size*, для этого используется двумерный массив логических значений *positions*, для работы с данным массивом реализованы методы *set_piece*, *update_next_position* и *update_max_piece_size*.

Метод *set_piece* получает в качестве аргументов двумерный массив логических значений и обрезок. Позициям, на которых расположен переданный обрезок, присваивается логическое значение «истина».

Метод *update_next_position* принимает в качестве агрумента двумерный массив логических значений. В данном методе построчно происходит просмотр всех позиций, начиная с позиции, записанной в поле *_next_unfilled_position*. Как только встречается позиция, имеющая логическое значение «ложь» (позиция, не занятая обрезком), ее координаты записываются в поле *_next_unfilled_position*. Если такой позиции найдено не было, то в поле *_next_unfilled_position* записываются координаты (*_size*, *_size*).

Метод *update_max_piece_size* принимает в качестве агрумента двумерный массив логических значений. В данном методе последовательно просматриваются пары позиций, находящиеся правее и ниже позиции,

записанной в поле *_next_unfilled_position*. Если в *k*-й паре хотя бы одна из позиций занята обрезком, то *k* — максимальный размер обрезка.

Для доступа к полям *_pieces* и *_max_piece_size* реализованы методы *get_pieces* и *get_max_piece_size*. Для плучения количества обрезков на столешнице реализован метод *get_number_of_pieces*. Чтобы узнать, есть ли на столешнице свободные позиции, реализован метод *is_filled*, если одна из координат позиции, записанной в поле *_next_unfilled_position* находится за пределами столешницы, то столешница является заполненной и данный метод возвращает «истину».

Основной алгоритм программы реализован в методах класса *Solver*.

Метод *solve* реализует основной алгоритм. В качестве аргумента данный метод принимает целое число *n* — длина стороны столешницы. Описание работы данного метода будет представлено после рассмотрения основных идей оптимизации данной задачи.

Рассмотрим решения поставленной задачи для малых значений *n*. На рисунке 1 изображены решения данной задачи для *n*, равных 2, 4, 6, 3 и 9. Заметим, что размещение обрезков на столешнице для заданного $n = a \cdot b$, где *a* — наименьший простой делитель числа *n*, совпадает с размещением обрезков в решении для столешницы со стороной размера *a*, а размер стороны обрезка умножен на *b*. Таким образом решение данной задачи для составного числа $n = a \cdot b$ можно свести к решению задачи для столешницы со стороной размера *a*, причем *a* — простое число.

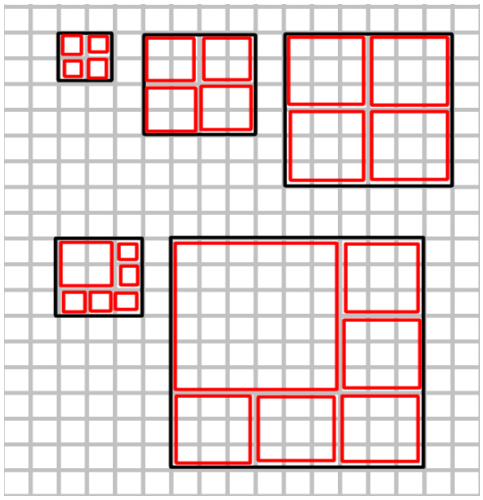

Рисунок 1 - Решение послабленной задачи для малых значений *N*.

Для описанного выше разложения числа *n* используется метод *factorize*. Метод *factorize* принимает в качестве аргумента целое число *n*, в качестве результата метод возвращает пару значений *a* и *b*, где *a* — наименьший простой делитель числа *n*, отличный от единицы, $b=\frac{n}{a}$.

Для того, чтобы ограничить дальнейший перебор вариантов, введена оценка максимального количества обрезков на столешнице. К примеру, если заполнить столешницу обрезками с длиной стороны 1, то максимальное количество обрезков будет равно n^2, данную оценку можно удучшить, если поместить в один из углов обрезок с длиной стороны $n-1$, а все оставшееся пространство заполнить обрезками с длиной стороны 1, в данном случае максимальное количество обрезков будет равно $2\cdot n$. Рассмотрим вариант, когда в один из углов столешницы вставляется обрезок с длиной стороны $\lfloor\frac{n+1}{2}\rfloor$, оставшееся пространство заполнено тремя обрезками с длиной стороны $n-\lfloor\frac{n+1}{2}\rfloor$, и, если отанется незаполненное пространство, оно будет заполнено обрезками с длиной стороны 1. При заполнении столешницы таким образом максимальное количество обрезков на ней вычисляется по следующей формуле:

$$k=2\cdot(\lfloor\frac{n+1}{2}\rfloor+1).$$

На рисунке 2 изображены описанные варианты заполнения столешницы для $n=7$.

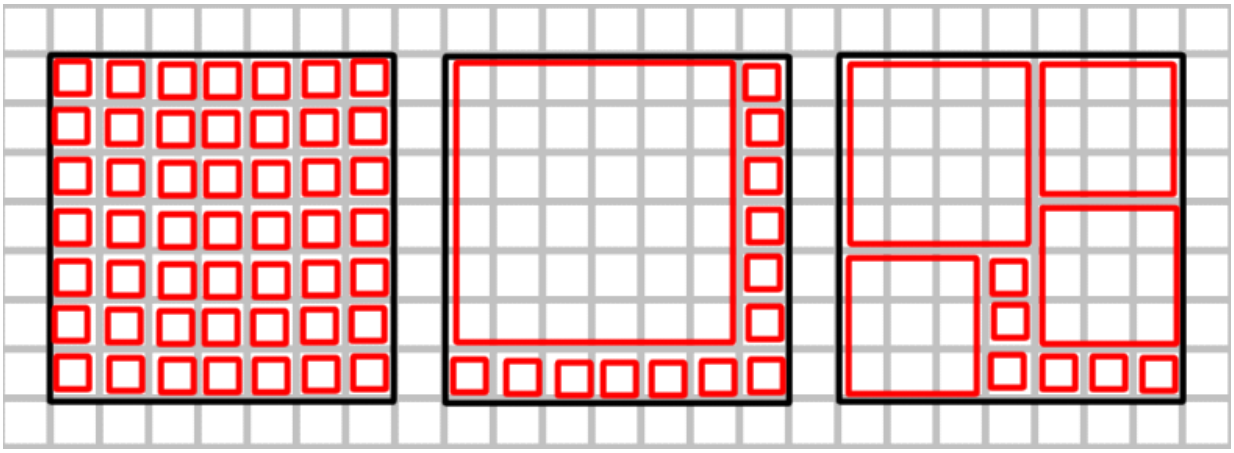

Рисунок 2 - Тривиальные варианты заполнения столешницы для $n=7$.

Метод *get_trivial_solution* принимает на вход незаполненный объект класса *Desk* и заполняет его в соответствии с третьим описанным вариантом заполнения.

Можно заметить, что решения данной задачи содержат в одном из углов столешницы (для определенности будем считать, что в левом верхнем) обрезок размера $k \geq \lfloor \frac{n+1}{2} \rfloor$, справа и снизу от которого находятся обрезки размера $n-k$. Таким образом в качестве начальных заполнений столешницы используются расстановки обрезков, такие, что в левом верхнем углу находится обрезок размера $\lfloor \frac{n+1}{2} \rfloor \leq k \leq 2$, справа и снизу от которого расположено $\lfloor \frac{n-k}{k} \rfloor$ обрезков размера $n-k$. На рисунке 3 изображен пример начальных расстановок обрезков для $n=11$.

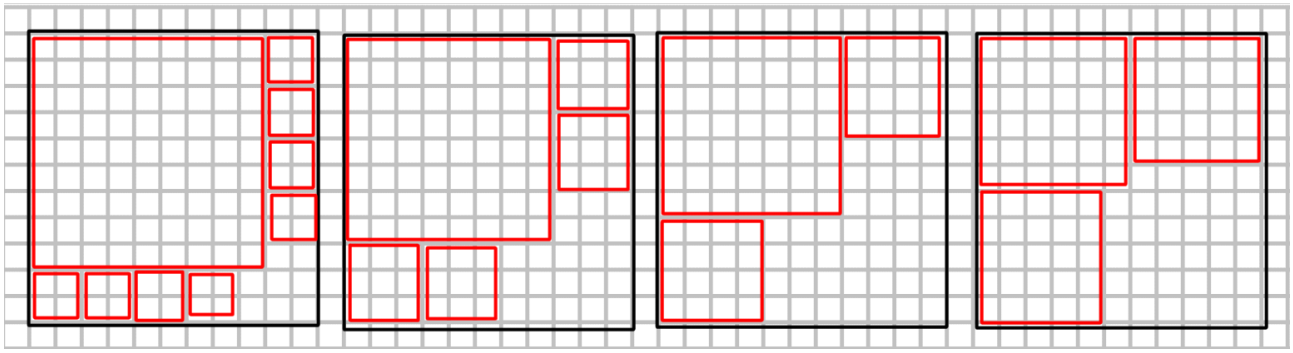

Рисунок 3 - Возможные начальные расстановки обрезков для $n=11$.

В реализованном алгоритме варианты расстановки обрезков на столешнице хранятся в стеке. Описанные выше начальные расстановки

обрезков добавляются в стек с помошью помощью метода *add_possible_start_positions.* В качестве аргумента данный метод принимает стек и добавляет в него начальные позиции, описанные выше, причем позиция с наименьшей длиной стороны наибольшего обрезка располагается на вершине стека.

Метод *solve* выполняет решение поставленной задачи. Переданное в качестве аргумента число *n* разделяется на множители, решение сначала находится для *a* — наименьшего простого делителя числа *n*, после нахождения решения размеры обрезков домножаются на $b=\frac{n}{a}$. В переменной *record* хранится минимальное найденное количество обрезков которыми можно заложить столешницу, данная переменная инициализируется значением $k=2\cdot(\lfloor\frac{n+1}{2}\rfloor+1)$. В переменной *solution* хранится расположение обрезков на столешнице, соответствующее записанному в переменной *record* значению, с помощью метода *get_trivial_solution* в переменную *solution* заносится расположение обрезков на столешнице, соответствующее начальному значению переменной *record*. С помощью стека будет производиться обход дерева возможных решений, с помощью метода *add_possible_start_positions* в стек заносятся возможные начальные расстановки обрезков на столешнице. Далее идет основной цикл, на каждой итерации которого извлекается и просматривается расположение обрезков на столешнице, находящейся на вершине стека. Если в текущем расположении отсутствуют свободные позиции и количество обрезков на столешнице меньше, чем значение, записанное в переменной *record*, то информация, записанная в переменных *record* и *solution* обновляется в соответствии с текущим расположением обрезков. Если в текущем расположении обрезков есть незанятые позиции, и при добавлении любого обрезка их количество на столешнице будет равно или будет превышать значение, записанное в переменной *record*, то происходит обработка следующего элемента из стека. Во всех остальных случаях в стек добавляются расстановки, получаемые из текущей расстановки добавлением обрезков всех

возможных размеров верхний левый угол которых будет занимать верхнюю левую свободную позицию на столешнице. Данный алгоритм продолжается, пока в стеке не останется элементов.

Выводы.

В ходе выполнения программы реализован алгоритм, выполняющий замощение квадрата квадратами меньшего размера. Для решения данной задачи реализован класс *Desk*, являющийся моделью квадрата, заполнение которого происходит в процессе работы алгоритма. Алгоритм основан на обходе в глубину дерева возможных расстановок квадратов.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <vector>
#include <stack>
#include <cmath>
#include <tuple>


/*
 * the piece is defined as the coordinates of its upper left corner
 * and the size of its side, these parameters are stored in the
tuple
 */
typedef std::tuple<unsigned, unsigned, unsigned> piece_t;
/*
 * position is defined as x and y coordinates, stored in pair
 */
typedef std::pair<unsigned, unsigned> position_t;


/*
 * class Desk describing the desk on which the pieces are placed
 */
class Desk
{
public:
    /*
     * desk is defined by the size of its side
     */
    explicit Desk(unsigned size);
    /*
     * copy constructor
     */
    Desk(const Desk &other);

    /*
     * adding a piece to the left top free position
     */
    void add_piece(unsigned size);
    /*
     * adding a piece to a given position
     */
    void add_piece(unsigned size, position_t position);
    /*
     * returns a vector containing the pieces located on the desk
     */
    std::vector<piece_t> get_pieces();
    /*
       * returns the maximum size of a piece's side that can be
placed in the top left free position
     */
```

```
        unsigned get_max_piece_size() const;
        /*
         * returns the number of pieces located on the desk
         */
        unsigned get_number_of_pieces() const;
        /*
           * returns true if there are no empty positions left on the
desk
         */
        bool is_filled() const;

    private:
        /*
         * desk side size
         */
        unsigned _size;
        /*
         * coordinates of the upper left free position
         */
        position_t _next_unfilled_position;
        /*
          * the maximum size of a piece that can be placed in the upper
left free position
         */
        unsigned _max_piece_size;
        /*
         * vector of pieces that located on the desk
         */
        std::vector<piece_t> _pieces;

        /*
          * methods needed to determine the top left free position and
the maximum size of a piece
          * that can be placed on it after adding the next piece to the
desk
         */
        void set_piece(bool positions[], piece_t &piece) const;
        void update_next_position(const bool positions[]);
        void update_max_piece_size(const bool positions[]);

    };


    Desk::Desk(unsigned size)
    {
        _size = size;
        _max_piece_size = _size - 1;
        _next_unfilled_position = {0, 0};
        _pieces.reserve(2 * ((_size + 1) / 2 + 1));
    }

    Desk::Desk(const Desk &other)
    {
        _size = other._size;
        _max_piece_size = other._max_piece_size;
        _next_unfilled_position = other._next_unfilled_position;

        _pieces.reserve(2 * ((_size + 1) / 2 + 1));
```

```
        for (auto &piece: other._pieces)
                                    _pieces.emplace_back(std::get<0>(piece),
std::get<1>(piece), std::get<2>(piece));
    }

    void Desk::add_piece(unsigned size)
    {
        add_piece(size, _next_unfilled_position);
    }

    void Desk::add_piece(unsigned size, position_t position)
    {
        _pieces.emplace_back(position.first, position.second, size);

        bool positions[_size][_size];
        for (unsigned i = 0; i < _size; i++)
            for (unsigned j = 0; j < _size; j++)
                positions[i][j] = false;

        for (auto &piece: _pieces)
            set_piece((bool *) positions, piece);

        update_next_position((bool *) positions);
        update_max_piece_size((bool *) positions);

    }

    std::vector<piece_t> Desk::get_pieces()
    {
        return _pieces;
    }

    void Desk::set_piece(bool positions[], piece_t &piece) const
    {
        unsigned position_x = std::get<0>(piece);
        unsigned position_y = std::get<1>(piece);
        unsigned piece_size = std::get<2>(piece);
        for (unsigned y = position_y; y < position_y + piece_size; y++)
             for (unsigned x = position_x; x < position_x + piece_size;
x++)
                positions[_size * y + x] = true;
    }

    void Desk::update_next_position(const bool positions[])
    {
        unsigned position_x = _next_unfilled_position.first;
        unsigned position_y = _next_unfilled_position.second;

         for (unsigned i = _size * position_y + position_x; i < _size *
_size; i++)
            if (!positions[i])
            {
                _next_unfilled_position = {i % _size, i / _size};
                return;
            }
        _next_unfilled_position = {_size, _size};
    }
```

```
    void Desk::update_max_piece_size(const bool positions[])
    {
        unsigned position_x = _next_unfilled_position.first;
        unsigned position_y = _next_unfilled_position.second;

         _max_piece_size = position_x + position_y == 0 ? _size - 1 :
std::min(_size - position_x, _size - position_y);
         for (unsigned i = 0; i < std::min(_size - position_x, _size -
position_y); i++)
               if (positions[_size * (position_y + i) + position_x] ||
positions[_size * position_y + position_x + i])
            {
                _max_piece_size = i;
                break;
            }
    }

    unsigned Desk::get_max_piece_size() const
    {
        return _max_piece_size;
    }

    unsigned Desk::get_number_of_pieces() const
    {
        return _pieces.size();
    }

    bool Desk::is_filled() const
    {
        unsigned position_x = _next_unfilled_position.first;
        unsigned position_y = _next_unfilled_position.second;
        return position_y == _size || position_x == _size;
    }


    /*
     * class Solver defines methods that solve the task
     */
    class Solver
    {
    public:
        /*
         * the main algorithm for solving the task
         */
        std::vector<piece_t> solve(unsigned n);

    private:
        /*
         * methods needed to optimize the algorithm
         */
        std::pair<unsigned, unsigned> factorize(unsigned n);
        void get_trivial_solution(Desk &desk, unsigned desk_size);
            void  add_possible_start_positions(std::stack<Desk>  &stack,
unsigned desk_size);

    };
```

```
    std::vector<piece_t> Solver::solve(unsigned n)
    {
        std::pair<unsigned, unsigned> multipliers = factorize(n);
        unsigned desk_size = multipliers.first;
        unsigned pieces_multiplier = multipliers.second;

        unsigned record = 2 * ((desk_size + 1) / 2 + 1);
        Desk solution(desk_size);
        get_trivial_solution(solution, desk_size);

        std::stack<Desk> stack;
        add_possible_start_positions(stack, desk_size);

        while (!stack.empty())
        {
            Desk &current_desk = stack.top();

                                    if    (current_desk.is_filled()    &&
current_desk.get_number_of_pieces() < record)
            {
                record = current_desk.get_number_of_pieces();
                solution = current_desk;
                stack.pop();
                continue;
            }

            if (current_desk.get_number_of_pieces() + 1 >= record)
            {
                stack.pop();
                continue;
            }

                                  for    (unsigned    i    =    2;    i    <=
current_desk.get_max_piece_size(); i++)
            {
                stack.emplace(current_desk);
                Desk &new_desk = stack.top();
                new_desk.add_piece(i);
            }
            current_desk.add_piece(1);

        }

        std::vector<piece_t> solution_pieces = solution.get_pieces();
        for (auto &piece: solution_pieces)
        {
            std::get<0>(piece) = std::get<0>(piece) * pieces_multiplier
+ 1;
            std::get<1>(piece) = std::get<1>(piece) * pieces_multiplier
+ 1;
            std::get<2>(piece) *= pieces_multiplier;
        }

        return solution_pieces;
    }

    std::pair<unsigned, unsigned> Solver::factorize(unsigned n)
    {
```

```
          for (unsigned divider = 2; divider <= unsigned(std::sqrt(n));
divider++)
              if (n % divider == 0)
                  return {divider, n / divider};
          return {n, 1};
      }

      void Solver::get_trivial_solution(Desk &desk, unsigned desk_size)
      {
          desk.add_piece((desk_size + 1) / 2);
          for (unsigned i = 0; i < 3; i++)
              desk.add_piece(desk_size / 2);
          while (!desk.is_filled())
              desk.add_piece(1);
      }

      void  Solver::add_possible_start_positions(std::stack<Desk>  &stack,
unsigned desk_size)
      {
          for (unsigned i = 2; i <= desk_size / 2; i++)
          {
              stack.emplace(desk_size);
              Desk &current_desk = stack.top();
              current_desk.add_piece(desk_size - i);
              for (unsigned j = 0; j < (desk_size - i) / i; j++)
              {
                  current_desk.add_piece(i, {desk_size - i, i * j});
                  current_desk.add_piece(i, {i * j, desk_size - i});
              }
          }
      }

      int main()
      {
          unsigned n;
          std::cin >> n;
          Solver solver;
          auto solution = solver.solve(n);

          std::cout << solution.size() << '\n';
          for (auto &piece: solution)
                          std::cout  <<  std::get<0>(piece)  <<  '  '  <<
std::get<1>(piece) << ' ' << std::get<2>(piece) << '\n';

          return 0;
      }
```