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. 2384 _____ Лавренова Ю.Д.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург

2024

Цель работы.

Изучить алгоритм поиска с возвратом. Применить полученные знания для решения задачи про квадрирование квадрата.

Задание.

Вариант 2р.

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 (см. рис. 1)

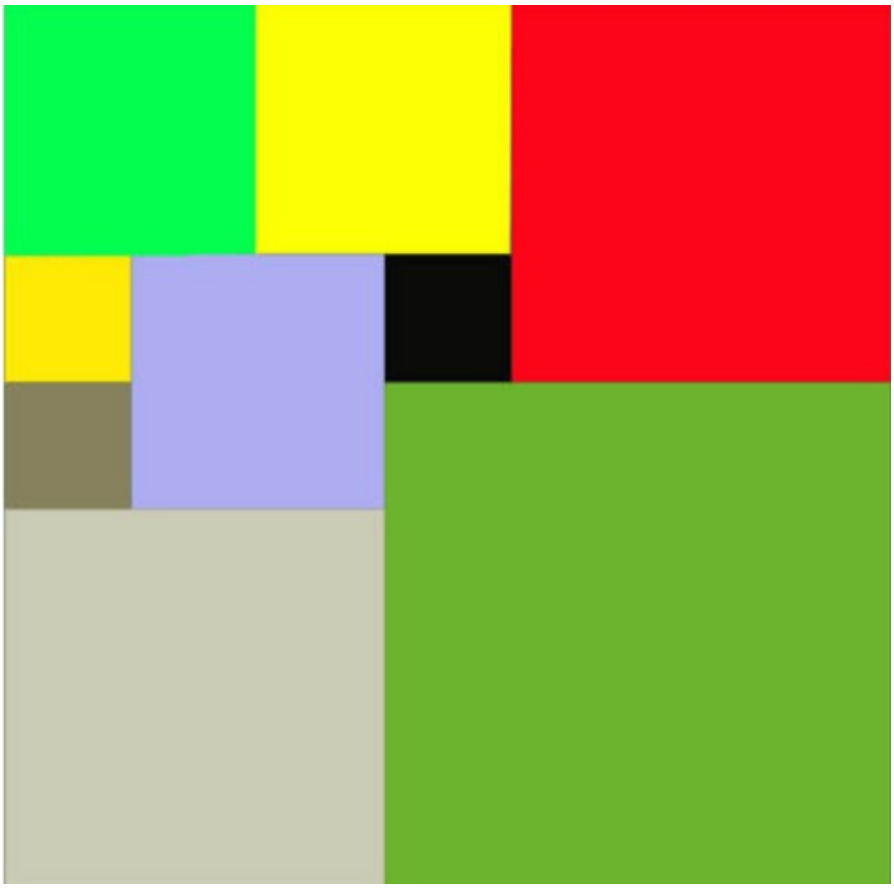

Рис. 1 – Оптимальное заполнение квадрата 7 на 7

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 <= 40.

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.

Методы оптимизации алгоритма.

Для успешного решения данной задачи использовались следующие оптимизации перебора с возвратом:

1) Если N – четное число, то ответ на задачу равен 4, поскольку это самый эффективный способ замостить квадрат с четной стороной.

2) В левый верхний угол ставится квадрат со стороной $n_1 = \lfloor \frac{N}{q} \rfloor * ¿ \lfloor \frac{q+1}{2} \rfloor$ $n_1 = \lfloor \frac{\frac{N}{q}+1}{2} \rfloor$, где N – сторона искомого квадрата, q – максимальный простой делитель числа N.

3) Под квадрат со стороной $n_2 = \lfloor \frac{\frac{N}{q}}{2} \rfloor$ ставится квадрат со стороной так, что он граничит с квадратом выше него и с левой границей столешницы. Правее квадрата со стороной n_1 ставится квадрат со стороной n_2 таким образом, что правая сторона меньшего квадрата примыкает к правой границе столешницы, а левая – к большому квадрату.

4) Если количество квадратов, входящих в текущий набор для составления столешницы, превышает наилучший результат, то данная ветвь возможного решения заканчивается.

5) Если площадь текущего набора квадратов равна площади столешницы, то происходит проверка на улучшение решения, используя минимальное количества квадратов. Вне зависимости от результата данная ветвь решения заканчивается.

Выполнение работы.

В ходе решения задачи реализован класс *Square*.

Класс *Square* представляет собой квадрат, который размещается на столешнице. Класс имеет конструктор, который принимает на вход *x,y* координаты левого верхнего угла квадрата и *size* - длину его стороны.

Реализованы дополнительные функции:

- *bool check_new_square(int x, int y, std::vector<Square>& table)* – функция проверки корректности вставки квадрата в столешницу по координатам его верхнего левого угла. Функция проходится по всем квадратам, которые уже вставлены, и проверяет пересечения с ними. Если координата левого верхнего угла нового квадрата заходит на площадь какого-либо уже вставленного, то функция возвращает *true*, иначе *false*.
- *void backtraking(std::vector<Square>& result, std::vector<Square> curr_table, int curr_area, int count, int min_x, int min_y, int& k, int n)* – основная функция, решающая поставленную задачу. Функция принимает на вход вектор, в котором будет храниться промежуточный результат (*result*), вектор с квадратами, которые уже помещены на столешницу (*curr_table*) (для данной ветви решения), текущую площадь, которую занимают выставленные квадраты (*curr_area*), количество выставленных квадратов (*count*), минимально возможную *x* и *y* координату левого верхнего угла нового квадрата (*min_x*, *min_y*), минимальное количество квадратов для решения задачи, размер исходного квадрата. С помощью вложенного цикла выполняется перебор возможных координат левого верхнего угла квадрата, далее так как при рекурсивном запуске бэктрекинга увеличивается только *y*, то есть область сужается только по вертикали считается значение *limit*, отвечающее за максимальный размер потенциального кандидата. Далее в столешницу вставляются квадраты всевозможных размеров от *limit* и менее и увеличивается занятая площадь. В случае, если заполненная площадь стала равна площади квадрата обновляется минимальное значение потраченных для этого квадратов

и запоминается результат в вектор. Если же количество затраченных квадратов уже превысило рекорд, а площадь еще не заполнена, то такая ветвь обрывается.

Исследование зависимости времени от размеров квадрата.

Приведено исследование времени работы алгоритмы на различных тестовых данных. Результаты исследования смотрите в таблице 3 и на рисунке 1.

Таблица 3. Зависимость времени от входных данных

Входные данные	Время,мс
2	0.1
3	0.2
4	0.2
5	0.21
6	0.2
7	0.64
8	0.2
9	0.6
10	0.2
11	0.774
12	0.2
13	2.362
14	0.2
15	0.6
16	0.2
17	11.214
18	0.2
19	32.942
20	0.3
21	0.6
22	0.1
23	112.805
24	0.3
25	0.19
26	0.1
27	0.5
28	0.1
29	759.807
30	0.2
31	1688.208
32	0.2
33	0.5
34	0.1
35	0.16
36	0.1

37	9012.994
38	0.2
39	0.5
40	0.1

Рисунок 1. Зависимость времени от входных данных

Анализируя график, можно сделать вывод о том, что в четных значениях *N* показатели времени максимально приближены к нулю, а также о том, что наибольшие показатели времени достигаются в наибольших простых числах (в нашем случае 31 и 37). Это связно с тем, что в таком случае не получается применить оптимизацию с заменой числа на его наименьший общий делитель.

Выводы.

Был изучен алгоритм поиска с возвратом, реализованы функции с использованием оптимизаций, которые решают поставленную задачу, проведено исследование времени выполнения поиска решения от размеров квадрата, проанализированы результаты исследования.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb1.cpp

```
#include <iostream>
#include <vector>


class Square{
    public:
        int x_coord;
        int y_coord;
        int size;
          Square(int x, int y, int size): x_coord{x}, y_coord{y},
size(size){}
};


void print_result(std::vector<Square>& table, int min){
    std::cout<<table.size()<<"\n";
    for(size_t i = 0; i < table.size(); i++){
                                  std::cout<<(table[i].x_coord*min)+1<<"
"<<(table[i].y_coord*min)+1<<" "<<table[i].size*min<<"\n";
    }
}

bool check_new_square(int x, int y, std::vector<Square>& table){
    for (size_t i = 0; i < table.size(); i++){
            if ((table[i].x_coord <= x && x < table[i].x_coord +
table[i].size) && (table[i].y_coord <= y && y < table[i].y_coord +
table[i].size)){
            return true;
        }
    }
    return false;
}

void backtraking(std::vector<Square>& result, std::vector<Square>
curr_table, int curr_area, int count, int min_x, int min_y, int& k, int
n){
    for (int x = min_x; x < n; x++){
        for (int y = min_y; y < n; y++){
            if(!check_new_square(x,y,curr_table)){
                int limit = std::min(n-x, n-y);
                for (int i = 0; i < curr_table.size(); i++){
                     if (curr_table[i].x_coord + curr_table[i].size
> x && curr_table[i].y_coord > y){
                                              limit = std::min(limit,
curr_table[i].y_coord - y);
                    }
                }

                for(int sq_size = limit; sq_size > 0; sq_size--){
                    Square new_square = Square(x,y,sq_size);
                    curr_table.push_back(new_square);

                     if (curr_area + new_square.size*new_square.size
== n*n){
```

```
                            if (count + 1 < k){
                                k = count + 1;
                                result = curr_table;
                            }
                        }
                        else {
                            if (count + 1 < k){
                                backtraking(result, curr_table,
curr_area + new_square.size*new_square.size, count + 1, x, y + sq_size,
k, n);
                            }
                        }
                        curr_table.pop_back();
                    }
                    return;

                }
            }
            min_y = n / 2;
        }
    }

    int main(){
        int n;
        std::cin>>n;
        int k;
        std::vector<Square> square_table;
        std::vector<Square> result;
        int min_simple = 1;

        if (n % 2 == 0){
            k = 4;
            square_table.push_back(Square(0,0,n/2));
            square_table.push_back(Square(n/2,0,n/2));
            square_table.push_back(Square(0,n/2,n/2));
            square_table.push_back(Square(n/2,n/2,n/2));
            result = square_table;
        }

        else{
            k = n * 2 + 1;
            for (int i = 2; i < n; i++){
                if (n % i == 0) min_simple = i;
            }

            n /= min_simple;
            square_table.push_back(Square(0,0,(n + 1)/2));
            square_table.push_back(Square(0,(n + 1)/2, n / 2));
            square_table.push_back(Square((n + 1)/2,0, n / 2));

              int curr_area = (((n + 1)/2)*((n + 1)/2) + 2 * (n/2) *
(n/2));
            int new_count = 3;
            int min_x = n / 2;
            int min_y = (n + 1)/2;
               backtraking(result, square_table, curr_area, new_count,
min_x, min_y, k, n);
        }
```

```
        print_result(result, min_simple);
        return 0;
    }
```

Название файла: test.cpp

```
    #include <iostream>
    #include <vector>
    #include <chrono>

    class Square{
        public:
            int x_coord;
            int y_coord;
            int size;
              Square(int x, int y, int size): x_coord{x}, y_coord{y},
size(size){}
    };


    void print_result(std::vector<Square>& table, int min){
        std::cout<<table.size()<<"\n";
        for(size_t i = 0; i < table.size(); i++){
                                    std::cout<<(table[i].x_coord*min)+1<<"
"<<(table[i].y_coord*min)+1<<" "<<table[i].size*min<<"\n";
        }
    }

    bool check_new_square(int x, int y, std::vector<Square>& table){
        for (size_t i = 0; i < table.size(); i++){
                if ((table[i].x_coord <= x && x < table[i].x_coord +
table[i].size) && (table[i].y_coord <= y && y < table[i].y_coord +
table[i].size)){
                    //std::cout<<"1"<<table[i].x_coord<<table[i].y_coord
<<" "<<x<<y<<table[i].y_coord + table[i].size<<"\n";
                return true;
            }
        }
        //std::cout<<"2"<<x<<y<<"\n";
        return false;
    }

    void backtraking(std::vector<Square>& result, std::vector<Square>
curr_table, int curr_area, int count, int min_x, int min_y, int& k, int
n){
        for (int x = min_x; x < n; x++){
            for (int y = min_y; y < n; y++){
                if(!check_new_square(x,y,curr_table)){
                    //print_result(curr_table);
                    int limit = std::min(n-x, n-y);
                    for (int i = 0; i < curr_table.size(); i++){
                         if (curr_table[i].x_coord + curr_table[i].size
> x && curr_table[i].y_coord > y){
                                                limit = std::min(limit,
curr_table[i].y_coord - y);
                        }
```

```
                    }
                    //std::cout<<limit<<"\n";
                    for(int sq_size = limit; sq_size > 0; sq_size--){
                        Square new_square = Square(x,y,sq_size);
                        curr_table.push_back(new_square);
                        //print_result(curr_table);
                        if (curr_area + new_square.size*new_square.size
== n*n){
                            if (count + 1 < k){
                                k = count + 1;
                                result = curr_table;
                            }
                        }
                        else {
                            if (count + 1 < k){
                                //std::cout<<k<<"k\n";
                                        backtraking(result, curr_table,
curr_area + new_square.size*new_square.size, count + 1, x, y + sq_size,
k, n);
                            }
                        }
                        curr_table.pop_back();
                    }
                    return;

                }
            }
            min_y = n / 2;
        }
    }


    int main(){
        for (int i = 2; i < 41; i++){
            int n = i;
            auto start = std::chrono::high_resolution_clock::now();

            int k;
            std::vector<Square> square_table;
            std::vector<Square> result;
            int min_simple = 1;

            if (n % 2 == 0){
                k = 4;
                square_table.push_back(Square(0,0,n/2));
                square_table.push_back(Square(n/2,0,n/2));
                square_table.push_back(Square(0,n/2,n/2));
                square_table.push_back(Square(n/2,n/2,n/2));
                result = square_table;
            }

            else{
                k = n * 2 + 1;
                for (int i = 2; i < n; i++){
                    if (n % i == 0) min_simple = i;
                }

                n /= min_simple;
```

```
                square_table.push_back(Square(0,0,(n + 1)/2));
                square_table.push_back(Square(0,(n + 1)/2, n / 2));
                square_table.push_back(Square((n + 1)/2,0, n / 2));

                 int curr_area = (((n + 1)/2)*((n + 1)/2) + 2 * (n/2) *
(n/2));
                int new_count = 3;
                int min_x = n / 2;
                int min_y = (n + 1)/2;
                 backtraking(result, square_table, curr_area, new_count,
min_x, min_y, k, n);
             }
                auto time = std::chrono::high_resolution_clock::now() -
start;
             std::cout<<i<< "    ";
              std::cout<<std::chrono::duration_cast<std::chrono::millisec
onds>(time).count() /*<< "ms "*/ << ".";
                                                                    std::cout<<
std::chrono::duration_cast<std::chrono::microseconds>(time).count()     -
std::chrono::duration_cast<std::chrono::milliseconds>(time).count()     *
1000 /*<< "mcs\n"*/;
             std::cout<<"\n";
         }
         return 0;
     }
```