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

Студентка гр. 1383 Федорова О.В

Преподаватель Токарев А.П.

Санкт-Петербург

2023

Цель работы.

Изучить принцип работы алгоритма поиска с возвратом. Решить с его помощью задачу

Основные теоретические положения.

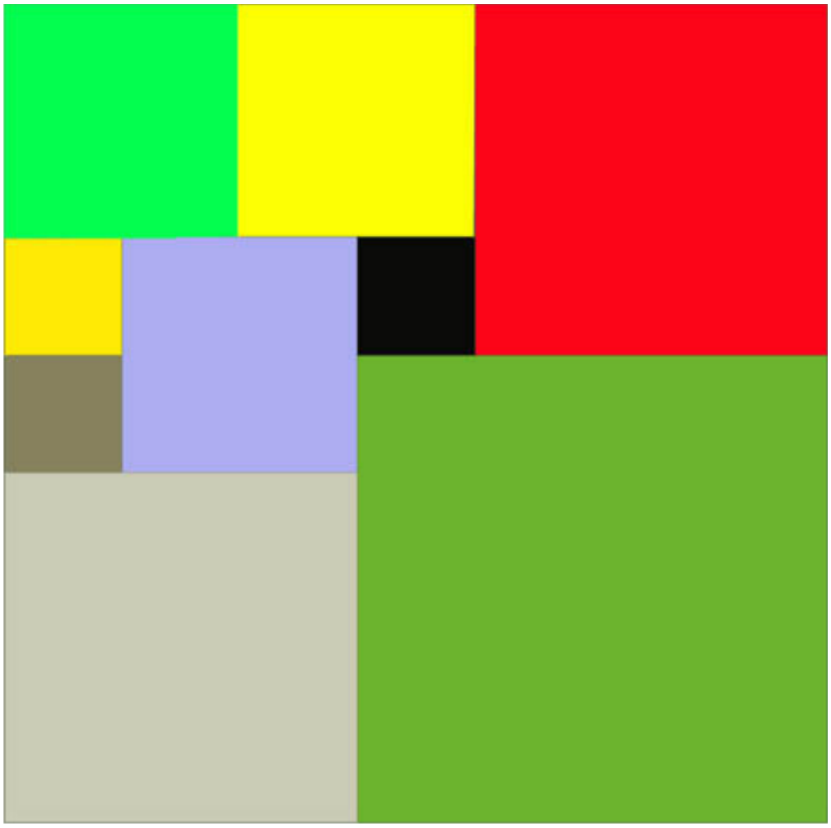
Поиск с возвратом, бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения. Метод поиска с возвратом является универсальным. Достаточно легко проектировать и программировать алгоритмы решения задач с использованием этого метода. Однако время нахождения решения может быть очень велико даже при небольших размерностях задачи (количестве исходных данных), причём настолько велико (может составлять годы или даже века), что о практическом применении не может быть и речи. Поэтому при проектировании таких алгоритмов, обязательно нужно теоретически оценивать время их работы на конкретных данных. Существуют также задачи выбора, для решения которых можно построить уникальные, «быстрые» алгоритмы, позволяющие быстро получить решение даже при больших размерностях задачи. Метод поиска с возвратом в таких задачах применять неэффективно.

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

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Вар. 1Р Решить все задания со Stepik

Реализация.

Описание алгоритма

Описана структура Rect, в которой в качестве полей выступали x,y координаты левого верхнего угла квадрата, size - размер квадрата, color - цвет. Решение задачи происходит в функции solve, в которой создаются начальные параметры для поиска оптимального решения задачи. Создаётся вектор r, в котором будет храниться набор квадратов, массив больших целых чисел - битовых масок, с помощью которой происходит запоминание самого квадрата. Число K и вектор ans - минимальное найденное число и вектор с оптимальным решением для квадратов. В программе происходит перебор всех решений и сравнение их с оптимальным.

Примененные оптимизации

1 Для ускорения программы вместо хранения двумерного массива чисел было принято решение хранить битовые маски. Использование чисел u_int_64_t позволяет решать задачу для N <= 64 и ускоряет процесс. Для матрицы размером N будет N чисел, каждое i число - маска соответствующей строки, при переводе числа в двоичную с-с у каждого числа в старшем бите будет 1, то есть изначально кадре число равняется 1(N+1), далее заполняются биты от младшего к старшему, что эквивалентно ячейкам в 0 - N столбцах. Для выяснения пересечения квадратов использовалась проверка на побитовое умножение битовой маски, если хоть одна строка в умножении давала не 1N+1 , значить произошла коллизия и квадрат поставить нельзя

2) первые 3 квадрата ставятся единым образом

3) если число составное, то его можно разложить так же, как и квадрат, размер которого равен наименьшему делителю

Разработанный программный код см. в приложении А.

Тестирование.

Проводится проверка на правильное размещение квадратов внутри большого квадрата и на количество использованных квадратов.

Результаты тестирования см. в приложении Б.

Таблица 1 – Зависимость количества операций от размера поля

Размер	Результат (к-во квадратов)
2	4
3	6
5	8
7	9
9	6
11	11
13	11
15	6
17	12
19	13
20	4

Выводы.

В результате работы была написана полностью рабочая программа решающая поставленную задачу при использовании изученных теоретических материалов. Программа была протестирована, результаты тестов удовлетворительны. По результатам исследования можем заключить, что зависимости числа операций от размера поля экспоненциальна.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include "vector"
typedef struct Rect{
    int x = 0;
    int y = 0;
    int size = 0;
    int color = 0;
    Rect(int x, int y, int size, int color):x(x), y(y), size(size), color(color){}
    Rect(const Rect & r2) {
        x = r2.x;
        y = r2.y;
        size = r2.size;
        color = r2.color;
    }
    Rect& operator = (const Rect & r2) = default;
    friend bool operator == (const Rect & r1, const Rect & r2) {
        return r1.x == r2.x && r1.y == r2.y && r1.size == r2.size;
    }
    friend bool operator > (const Rect & r1, const Rect & r2) {
        if(r1.y > r2.y) return true;
        if(r1.y == r2.y && r1.x > r2.x) return true;
        return false;
    }
    friend bool operator >= (const Rect & r1, const Rect & r2) {
        if(r1.y >= r2.y) return true;
        if(r1.y == r2.y && r1.x >= r2.x) return true;
        return false;
    }
    friend bool operator < (const Rect & r1, const Rect & r2) {
        if(r1.y < r2.y) return true;
        if(r1.y == r2.y && r1.x < r2.x) return true;
        return false;
    }
    friend bool operator <= (const Rect & r1, const Rect & r2) {
        if(r1.y <= r2.y) return true;
        if(r1.y == r2.y && r1.x <= r2.x) return true;
        return false;
    }

    friend Rect operator * (const Rect & r, int k) {
        return Rect(r.x * k, r.y * k, r.size * k, r.color);
    }

}Rect;
```

```
void multiply_vec(std::vector <Rect> & r, int k) {
    for(int i = 0; i < r.size(); i++) {
        r[i] = r[i] * k;
    }
}

int prime_del(const int & n) {
    std::vector <int> prime = {2,3,5,7,11,13, 17,19};
    for(int i = 0; i < prime.size(); i++) {
        if(n % prime[i] == 0 && n != prime[i]) return prime[i];
    }
    return n;
}


void print_rect(std::vector <Rect> ans) {
    for(int i = 0; i < ans.size(); i++) {
        std::cout<<ans[i].y + 1 <<" "<<ans[i].x + 1<<" "<<ans[i].size<<std::endl;
    }
}

void print_rect(std::vector <Rect> ans, const int & N) {
    std::vector <std::vector <int> > table(N);
    for(int i = 0; i < N; i++) {
        table[i] = std::vector<int>(N);
    }
    for(int i = 0; i < ans.size(); i++) {
        for(int y = 0; y < ans[i].size; y++) {
            for(int x = 0; x < ans[i].size; x++) {
                table[y + ans[i].y][x + ans[i].x] = ans[i].color;
            }
        }
    }
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            std::cout<<table[i][j]<<" ";
        }
        std::cout<<std::endl;
    }
}

void print_mask(std::vector <u_int64_t > bit_mask, const int & N) {
    for(int i = 0; i < N; i++) {
        while (bit_mask[i] > 1) {
            std::cout<<bit_mask[i] % 2<<" ";
            bit_mask[i] /= 2;
        }
        std::cout<<std::endl;
    }
```

```
}


bool free_rect(const std::vector <u_int64_t> & bit_mask, Rect r,
               const int & N, const std::vector <u_int64_t> & bins) {
    u_int64_t bit = (bins[r.size] - 1) * bins[r.x];
    for(int y = r.y; y < r.y + r.size; y++) {
        if(bit_mask[y] == bins[N] )return true;
        if((bit_mask[y] & bit) != 0) return false;
    }
    return true;
}

void fill_rect(std::vector <Rect> & rects, std::vector <u_int64_t> & bit_mask, const Rect &
r,
               int & count, const std::vector <u_int64_t> & bins) {
    rects.push_back(r);
    for(int y = r.y; y < r.y + r.size; y++) {
        bit_mask[y] += (bins[r.size] - 1) * bins[r.x];
    }
    count += r.size * r.size;
}

void find_free_position(int & i, int & j, int & count_free_in_line,const std::vector
<u_int64_t> & bit_mask,
                        const int & N, const std::vector <u_int64_t> & bins) {
    while(bit_mask[i] == bins[N+1] - 1)
        i++;
    u_int64_t tmp = bit_mask[i];
    while (tmp % 2 == 1) {
        j++;
        tmp/=2;
    }
    count_free_in_line = 0;
    while (tmp % 2 == 0) {
        count_free_in_line++;
        tmp/=2;
    }
}

void step(std::vector <Rect> rects, const int & N, int color, int count,
          int x_next, int y_next, int size, std::vector <u_int64_t> bit_mask,
          const std::vector <u_int64_t> & bins, std::vector <Rect> & ans, int & K) {
    Rect r(x_next, y_next, size, color + 1);
    if ( K > color + 1 && count < N * N && size > 0
         && !(count + size * size  < N * N && K == color + 2)
         && free_rect(bit_mask, r, N, bins)) {
        fill_rect(rects, bit_mask, r, count, bins);
        if(count < N * N) {
```

```
            int i = y_next, j = 0, count_free_in_line = 0;
            find_free_position(i, j, count_free_in_line, bit_mask, N, bins);
            for (int size_ = std::min(N - i, count_free_in_line); size_ > 0 ; size_--) {
                step(rects, N, color + 1, count, j, i, size_, bit_mask, bins, ans, K);
            }
        } else {
            if ( K > color + 1) {
                ans = rects;
                K = color + 1;
            }
        }
    }
}

std::vector <u_int64_t> create_bit_mask(const int & N, const std::vector<u_int64_t> &
bins) {
    std::vector <u_int64_t> bit_mask(N);
    for(int r = 0; r < N; r++) {
        bit_mask[r] = bins[N];
    }
    return bit_mask;
}

void start_fill_rect(const int & N, std::vector <u_int64_t> & bit_mask,
                     std::vector <Rect> & r, const std::vector <u_int64_t> & bins) {
    Rect first(0, 0, (N + 1)/2, 1);
    Rect second((N + 1)/2, 0, N/2, 2);
    Rect third(0, (N + 1)/2, N/2, 3);
    r.push_back(first);
    r.push_back(second);
    r.push_back(third);
    for(int y = 0; y < N/2; y++) {
        bit_mask[y] += bins[N] - 1;
    }
    for(int y = (N + 1)/2; y < N; y++) {
        bit_mask[y] += bins[N/2] - 1;
    }
    if(N % 2 != 0)
        bit_mask[N/2] += bins[(N+1)/2] - 1;
}

void solve(int N, const std::vector <u_int64_t> & bins, std::vector <Rect> & ans, int & K) {
    std::vector <Rect> r;
    std::vector <u_int64_t> bit_mask;
    int del = prime_del(N);
    int k = N / del;
    N = del;
    K = 2 * N + 1;
    bit_mask = create_bit_mask(del, bins);
```

```
    start_fill_rect(N, bit_mask, r, bins);
    for (int i = N - (N + 1)/2; i >= 1; i--) {
        step(r, N, 3,
            2 * ((int)N/2) * ((int)N/2) +  (int)((N + 1)/2) * (int)((N + 1)/2),
            (N + 1)/2, (N)/2, i, bit_mask,
            bins, ans, K);
    }
    multiply_vec(ans, k);
}

void start(int & N) {
    int K = 0;
    std::vector <Rect> ans;
    std::vector <u_int64_t> bins;
    u_int64_t r = 1;
    for(int i = 0; i < N + 2; i++, r*=2) {
        bins.push_back(r);
    }
    solve(N, bins, ans, K);
    std::cout << K << std::endl;
    print_rect(ans);
}

int main() {
    int N;
    std::cin>>N;
    start(N);
    return 0;
}
```

```
#include <iostream>
#include "vector"
typedef struct Rect{
    int x = 0;
    int y = 0;
    int size = 0;
    int color = 0;
    Rect(int x, int y, int size, int color):x(x), y(y), size(size), color(color){}
    Rect(const Rect & r2) {
        x = r2.x;
        y = r2.y;
        size = r2.size;
        color = r2.color;
    }
    Rect& operator = (const Rect & r2) = default;
    friend bool operator == (const Rect & r1, const Rect & r2) {
        return r1.x == r2.x && r1.y == r2.y && r1.size == r2.size;
    }
    friend bool operator > (const Rect & r1, const Rect & r2) {
```

```
        if(r1.y > r2.y) return true;
        if(r1.y == r2.y && r1.x > r2.x) return true;
        return false;
    }
    friend bool operator >= (const Rect & r1, const Rect & r2) {
        if(r1.y >= r2.y) return true;
        if(r1.y == r2.y && r1.x >= r2.x) return true;
        return false;
    }
    friend bool operator < (const Rect & r1, const Rect & r2) {
        if(r1.y < r2.y) return true;
        if(r1.y == r2.y && r1.x < r2.x) return true;
        return false;
    }
    friend bool operator <= (const Rect & r1, const Rect & r2) {
        if(r1.y <= r2.y) return true;
        if(r1.y == r2.y && r1.x <= r2.x) return true;
        return false;
    }

    friend Rect operator * (const Rect & r, int k) {
        return Rect(r.x * k, r.y * k, r.size * k, r.color);
    }

}Rect;

void multiply_vec(std::vector <Rect> & r, int k) {
    for(int i = 0; i < r.size(); i++) {
        r[i] = r[i] * k;
    }
}

int prime_del(const int & n) {
    std::vector <int> prime = {2,3,5,7,11,13, 17,19};
    for(int i = 0; i < prime.size(); i++) {
        if(n % prime[i] == 0 && n != prime[i]) return prime[i];
    }
    return n;
}


void print_rect(std::vector <Rect> ans) {
    for(int i = 0; i < ans.size(); i++) {
        std::cout<<ans[i].y + 1 <<" "<<ans[i].x + 1<<" "<<ans[i].size<<std::endl;
    }
}

void print_rect(std::vector <Rect> ans, const int & N) {
    std::vector <std::vector <int> > table(N);
```

```
    for(int i = 0; i < N; i++) {
        table[i] = std::vector<int>(N);
    }
    for(int i = 0; i < ans.size(); i++) {
        for(int y = 0; y < ans[i].size; y++) {
            for(int x = 0; x < ans[i].size; x++) {
                table[y + ans[i].y][x + ans[i].x] = ans[i].color;
            }
        }
    }
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            std::cout<<table[i][j]<<" ";
        }
        std::cout<<std::endl;
    }
}

void print_mask(std::vector <u_int64_t > bit_mask, const int & N) {
    for(int i = 0; i < N; i++) {
        while (bit_mask[i] > 1) {
            std::cout<<bit_mask[i] % 2<<" ";
            bit_mask[i] /= 2;
        }
        std::cout<<std::endl;
    }
}


bool free_rect(const std::vector <u_int64_t> & bit_mask, Rect r,
               const int & N, const std::vector <u_int64_t> & bins) {
    u_int64_t bit = (bins[r.size] - 1) * bins[r.x];
    for(int y = r.y; y < r.y + r.size; y++) {
        if(bit_mask[y] == bins[N] )return true;
        if((bit_mask[y] & bit) != 0) return false;
    }
    return true;
}

void fill_rect(std::vector <Rect> & rects, std::vector <u_int64_t> & bit_mask, const Rect &
r,
               int & count, const std::vector <u_int64_t> & bins) {
    rects.push_back(r);
    for(int y = r.y; y < r.y + r.size; y++) {
        bit_mask[y] += (bins[r.size] - 1) * bins[r.x];
    }
    count += r.size * r.size;
}
```

```
void find_free_position(int & i, int & j, int & count_free_in_line,const std::vector
<u_int64_t> & bit_mask,
                    const int & N, const std::vector <u_int64_t> & bins) {
   while(bit_mask[i] == bins[N+1] - 1)
      i++;
   u_int64_t tmp = bit_mask[i];
   while (tmp % 2 == 1) {
      j++;
      tmp/=2;
   }
   count_free_in_line = 0;
   while (tmp % 2 == 0) {
      count_free_in_line++;
      tmp/=2;
   }
}

void step(std::vector <Rect> rects, const int & N, int color, int count,
        int x_next, int y_next, int size, std::vector <u_int64_t> bit_mask,
        const std::vector <u_int64_t> & bins, std::vector <Rect> & ans, int & K) {
   Rect r(x_next, y_next, size, color + 1);
   if ( K > color + 1 && count < N * N && size > 0
       && !(count + size * size  < N * N && K == color + 2)
       && free_rect(bit_mask, r, N, bins)) {
      fill_rect(rects, bit_mask, r, count, bins);
      if(count < N * N) {
         int i = y_next, j = 0, count_free_in_line = 0;
         find_free_position(i, j, count_free_in_line, bit_mask, N, bins);
         for (int size_ = std::min(N - i, count_free_in_line); size_ > 0 ; size_--) {
            step(rects, N, color + 1, count, j, i, size_, bit_mask, bins, ans, K);
         }
      } else {
         if ( K > color + 1) {
            ans = rects;
            K = color + 1;
         }
      }
   }
}

std::vector <u_int64_t> create_bit_mask(const int & N, const std::vector<u_int64_t> &
bins) {
   std::vector <u_int64_t> bit_mask(N);
   for(int r = 0; r < N; r++) {
      bit_mask[r] = bins[N];
   }
   return bit_mask;
}
```

```
void start_fill_rect(const int & N, std::vector <u_int64_t> & bit_mask,
                     std::vector <Rect> & r, const std::vector <u_int64_t> & bins) {
   Rect first(0, 0, (N + 1)/2, 1);
   Rect second((N + 1)/2, 0, N/2, 2);
   Rect third(0, (N + 1)/2, N/2, 3);
   r.push_back(first);
   r.push_back(second);
   r.push_back(third);
   for(int y = 0; y < N/2; y++) {
      bit_mask[y] += bins[N] - 1;
   }
   for(int y = (N + 1)/2; y < N; y++) {
      bit_mask[y] += bins[N/2] - 1;
   }
   if(N % 2 != 0)
      bit_mask[N/2] += bins[(N+1)/2] - 1;
}

void solve(int N, const std::vector <u_int64_t> & bins, std::vector <Rect> & ans, int & K) {
   std::vector <Rect> r;
   std::vector <u_int64_t> bit_mask;
   int del = prime_del(N);
   int k = N / del;
   N = del;
   K = 2 * N + 1;
   bit_mask = create_bit_mask(del, bins);
   start_fill_rect(N, bit_mask, r, bins);
   for (int i = N - (N + 1)/2; i >= 1; i--) {
      step(r, N, 3,
         2 * ((int)N/2) * ((int)N/2) +  (int)((N + 1)/2) * (int)((N + 1)/2),
         (N + 1)/2, (N)/2, i, bit_mask,
         bins, ans, K);
   }
   multiply_vec(ans, k);
}

void start(int & N) {
   int K = 0;
   std::vector <Rect> ans;
   std::vector <u_int64_t> bins;
   u_int64_t r = 1;
   for(int i = 0; i < N + 2; i++, r*=2) {
      bins.push_back(r);
   }
   solve(N, bins, ans, K);
   std::cout << K << std::endl;
   print_rect(ans);
}
```

```cpp
int main() {
    int N;
    std::cin>>N;
    start(N);
    return 0;
}
#include <iostream>
#include "vector"
typedef struct Rect{
    int x = 0;
    int y = 0;
    int size = 0;
    int color = 0;
    Rect(int x, int y, int size, int color):x(x), y(y), size(size), color(color){}
    Rect(const Rect & r2) {
        x = r2.x;
        y = r2.y;
        size = r2.size;
        color = r2.color;
    }
    Rect& operator = (const Rect & r2) = default;
    friend bool operator == (const Rect & r1, const Rect & r2) {
        return r1.x == r2.x && r1.y == r2.y && r1.size == r2.size;
    }
    friend bool operator > (const Rect & r1, const Rect & r2) {
        if(r1.y > r2.y) return true;
        if(r1.y == r2.y && r1.x > r2.x) return true;
        return false;
    }
    friend bool operator >= (const Rect & r1, const Rect & r2) {
        if(r1.y >= r2.y) return true;
        if(r1.y == r2.y && r1.x >= r2.x) return true;
        return false;
    }
    friend bool operator < (const Rect & r1, const Rect & r2) {
        if(r1.y < r2.y) return true;
        if(r1.y == r2.y && r1.x < r2.x) return true;
        return false;
    }
    friend bool operator <= (const Rect & r1, const Rect & r2) {
        if(r1.y <= r2.y) return true;
        if(r1.y == r2.y && r1.x <= r2.x) return true;
        return false;
    }

    friend Rect operator * (const Rect & r, int k) {
        return Rect(r.x * k, r.y * k, r.size * k, r.color);
    }
```

```
}Rect;

void multiply_vec(std::vector <Rect> & r, int k) {
    for(int i = 0; i < r.size(); i++) {
        r[i] = r[i] * k;
    }
}

int prime_del(const int & n) {
    std::vector <int> prime = {2,3,5,7,11,13, 17,19};
    for(int i = 0; i < prime.size(); i++) {
        if(n % prime[i] == 0 && n != prime[i]) return prime[i];
    }
    return n;
}



void print_rect(std::vector <Rect> ans) {
    for(int i = 0; i < ans.size(); i++) {
        std::cout<<ans[i].y + 1 <<" "<<ans[i].x + 1<<" "<<ans[i].size<<std::endl;
    }
}

void print_rect(std::vector <Rect> ans, const int & N) {
    std::vector <std::vector <int> > table(N);
    for(int i = 0; i < N; i++) {
        table[i] = std::vector<int>(N);
    }
    for(int i = 0; i < ans.size(); i++) {
        for(int y = 0; y < ans[i].size; y++) {
            for(int x = 0; x < ans[i].size; x++) {
                table[y + ans[i].y][x + ans[i].x] = ans[i].color;
            }
        }
    }
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            std::cout<<table[i][j]<<" ";
        }
        std::cout<<std::endl;
    }
}

void print_mask(std::vector <u_int64_t > bit_mask, const int & N) {
    for(int i = 0; i < N; i++) {
        while (bit_mask[i] > 1) {
            std::cout<<bit_mask[i] % 2<<" ";
            bit_mask[i] /= 2;
        }
```

```
        std::cout<<std::endl;
    }
}



bool free_rect(const std::vector <u_int64_t> & bit_mask, Rect r,
               const int & N, const std::vector <u_int64_t> & bins) {
    u_int64_t bit = (bins[r.size] - 1) * bins[r.x];
    for(int y = r.y; y < r.y + r.size; y++) {
        if(bit_mask[y] == bins[N] )return true;
        if((bit_mask[y] & bit) != 0) return false;
    }
    return true;
}

void fill_rect(std::vector <Rect> & rects, std::vector <u_int64_t> & bit_mask, const Rect &
r,
               int & count, const std::vector <u_int64_t> & bins) {
    rects.push_back(r);
    for(int y = r.y; y < r.y + r.size; y++) {
        bit_mask[y] += (bins[r.size] - 1) * bins[r.x];
    }
    count += r.size * r.size;
}

void find_free_position(int & i, int & j, int & count_free_in_line,const std::vector
<u_int64_t> & bit_mask,
                        const int & N, const std::vector <u_int64_t> & bins) {
    while(bit_mask[i] == bins[N+1] - 1)
        i++;
    u_int64_t tmp = bit_mask[i];
    while (tmp % 2 == 1) {
        j++;
        tmp/=2;
    }
    count_free_in_line = 0;
    while (tmp % 2 == 0) {
        count_free_in_line++;
        tmp/=2;
    }
}

void step(std::vector <Rect> rects, const int & N, int color, int count,
          int x_next, int y_next, int size, std::vector <u_int64_t> bit_mask,
          const std::vector <u_int64_t> & bins, std::vector <Rect> & ans, int & K) {
    Rect r(x_next, y_next, size, color + 1);
    if ( K > color + 1 && count < N * N && size > 0
         && !(count + size * size  < N * N && K == color + 2)
         && free_rect(bit_mask, r, N, bins)) {
```

```
        fill_rect(rects, bit_mask, r, count, bins);
        if(count < N * N) {
            int i = y_next, j = 0, count_free_in_line = 0;
            find_free_position(i, j, count_free_in_line, bit_mask, N, bins);
            for (int size_ = std::min(N - i, count_free_in_line); size_ > 0 ; size_--) {
                step(rects, N, color + 1, count, j, i, size_, bit_mask, bins, ans, K);
            }
        } else {
            if ( K > color + 1) {
                ans = rects;
                K = color + 1;
            }
        }
    }
}

std::vector <u_int64_t> create_bit_mask(const int & N, const std::vector<u_int64_t> &
bins) {
    std::vector <u_int64_t> bit_mask(N);
    for(int r = 0; r < N; r++) {
        bit_mask[r] = bins[N];
    }
    return bit_mask;
}

void start_fill_rect(const int & N, std::vector <u_int64_t> & bit_mask,
                     std::vector <Rect> & r, const std::vector <u_int64_t> & bins) {
    Rect first(0, 0, (N + 1)/2, 1);
    Rect second((N + 1)/2, 0, N/2, 2);
    Rect third(0, (N + 1)/2, N/2, 3);
    r.push_back(first);
    r.push_back(second);
    r.push_back(third);
    for(int y = 0; y < N/2; y++) {
        bit_mask[y] += bins[N] - 1;
    }
    for(int y = (N + 1)/2; y < N; y++) {
        bit_mask[y] += bins[N/2] - 1;
    }
    if(N % 2 != 0)
        bit_mask[N/2] += bins[(N+1)/2] - 1;
}

void solve(int N, const std::vector <u_int64_t> & bins, std::vector <Rect> & ans, int & K) {
    std::vector <Rect> r;
    std::vector <u_int64_t> bit_mask;
    int del = prime_del(N);
    int k = N / del;
    N = del;
```

```
    K = 2 * N + 1;
    bit_mask = create_bit_mask(del, bins);
    start_fill_rect(N, bit_mask, r, bins);
    for (int i = N - (N + 1)/2; i >= 1; i--) {
        step(r, N, 3,
            2 * ((int)N/2) * ((int)N/2) +  (int)((N + 1)/2) * (int)((N + 1)/2),
            (N + 1)/2, (N)/2, i, bit_mask,
            bins, ans, K);
    }
    multiply_vec(ans, k);
}

void start(int & N) {
    int K = 0;
    std::vector <Rect> ans;
    std::vector <u_int64_t> bins;
    u_int64_t r = 1;
    for(int i = 0; i < N + 2; i++, r*=2) {
        bins.push_back(r);
    }
    solve(N, bins, ans, K);
    std::cout << K << std::endl;
    print_rect(ans);
}

int main() {
    int N;
    std::cin>>N;
    start(N);
    return 0;
}
```