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
ВАРИАНТ 4Р

Студент гр. 8303 _____ Хохлов Г.О.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

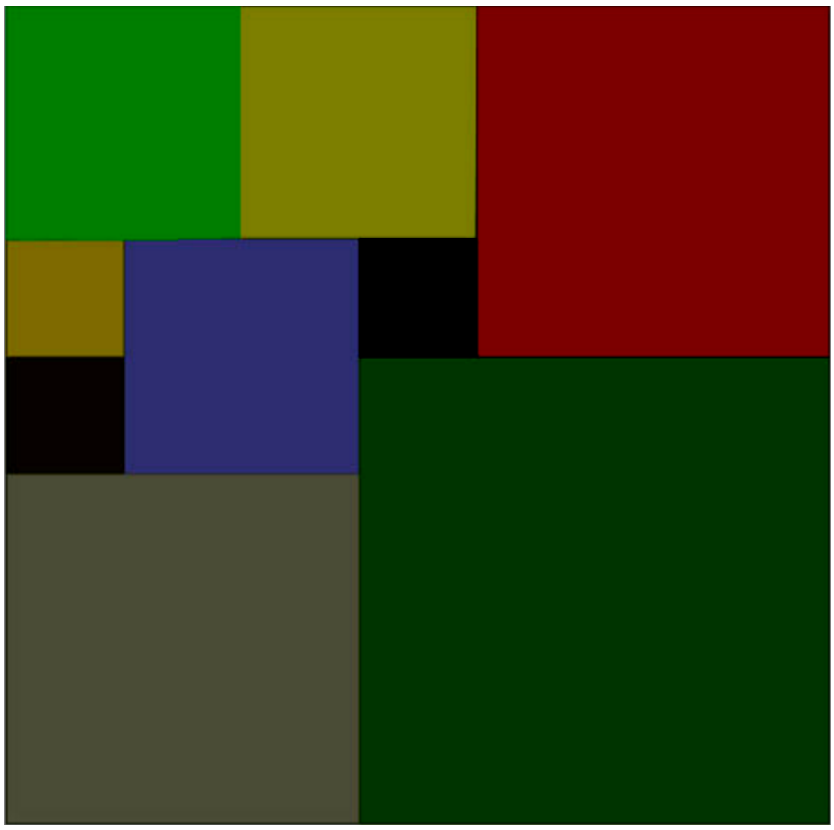
Цель работы.

Изучение алгоритма 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).

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 ($1 \le x, y \le N$) и длину стороны соответствующего обрезка (квадрата).

Пример входных данных

7

Соответствующие выходные данные

9
1 1 2
1 3 2
3 1 1
4 1 1
3 2 2
5 1 3
4 4 4
1 5 3
3 4 1

Индивидуализация.

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

Описание алгоритма.

Работа программы основана на алгоритме поиска с возвратом. Он заключается в полном переборе всех возможных вариантов заполнения прямоугольника квадратами.

Изначально вызывается рекурсивная функция вставки квадратов со сторонами от min(n, m)-1 до 1, где n, m — высота и ширина прямоугольника соответственно. И после каждой вставки опять рекурсивно вызывается функция вставки следующего квадрата. Выход из рекурсии происходит тогда, когда на поле нет пустых ячеек.

Для хранения решения используется двумерный вектор поля (vector<vector<int>> rect) и вектор вставленных квадратов (vector<InsertedRect> result). При откате отменяется вставка квадрата находящегося в конце вектора, это будет последний вставленный квадрат.

На каждом шаге мы пытаемся вставить квадраты от min(n, m)-1 до 1.

И максимальная глубина рекурсии будет n*m для заполнения поля только единичными квадратами. Тогда имеем $(\min(n,m)-1)^{n*m}$ вызовов рекурсивной функции и получаем сложность по количеству операций $O(\min(n,m)^{n*m})$.

На протяжении всей программы используется двумерный вектор поля и вектор для хранения результата разложения. Тогда сложность по памяти составляет O(n*m).

Использованные оптимизации.

Не имеет смысла долго искать координаты, по которым мы можем вставить текущий квадрат. Алгоритм находит первую свободную координату, и если в нее поместить квадрат нельзя, переходим к следующему квадрату. Таким образом можно избавиться от дублирования вариантов размещения.

Если обнаружено, что текущее число размещенных квадратов равно лучшему на данный момент размещению, а поле еще не полностью покрыто, происходит откат.

Для квадратов можно применить следующую оптимизацию. Нетрудно заметить, что два квадрата, размер одного из которых является наименьшим делимым (больше 1) размера другого, то такие квадраты имеют одинаковое минимальное разбиение (с учетом масштаба) и одинаковое число вариантов покрытия минимальным числом квадратов. Тогда для введенного квадрата ищем такое минимальное (больше 1) делимое, и заменяем размер квадрата. По результату работы программы нужно домножить координаты вставленных квадратов на отношение размеров исходного и заменяющего квадратов.

Описание структур данных.

Структура InsertedRect:

int x, y — координаты квадрата.

int size — размер квадрата.

Используется для хранения размещенного на поле квадрата.

vector<vector<int>> rect:

Хранит состояние поля в виде матрицы.

vector<InsertedRect> result:

Вектор для хранения размещенных квадратов.

Описание функций

void clearRect(vector<vector<int>> &_rect)

&_rect — ссылка на поле, представленное в виде матрицы.

Ничего не возвращает, используется для инициализации поля нулями.

void printRect(vector<vector<int>>& _rect)

&_rect — ссылка на поле, представленное в виде матрицы.

Ничего не возвращает, выводит текущее состояние поля в стандартный поток вывода.

void recovery(vector<vector<int>>& _rect, vector<InsertedRect> &_res)

&_rect — ссылка на поле, представленное в виде матрицы.

&_res — ссылка на вектор, в котором хранятся размещенные на поле квадраты.

Ничего не возвращает, производит откат последнего изменения поля и вектора результата.

bool isPushable(vector<vector<int>> &_rect, int x, int y, int size)

&_rect — ссылка на поле, представленное в виде матрицы.

int x,y — координаты квадрата

size — размер квадрата.

Возвращает true, если по координатам x,y можно вставить квадрат размера size, в ином случае возвращает false.

void push(vector<vector<int>>& _rect, int x, int y, int size)

&_rect — ссылка на поле, представленное в виде матрицы.

int x,y — координаты квадрата

size — размер квадрата.

Ничего не возвращает, размещает квадрат размера size по координатам x,y.

void recBack(vector<vector<int>>& _rect, int curSquare, int unPainted, int countOfSquares, vector<InsertedRect> &_res)

&_rect — ссылка на поле, представленное в виде матрицы.

curSquare — размер квадрата, который мы пытаемся вставить на текущем шаге.

&_res — ссылка на вектор, в котором хранятся размещенные на поле квадраты.

Тестирование.

```
Enter rectangle height&width:4 5
```

```
Minimal number of squares: 6
1 1 3
1 4 2
3 4 2
4 1 1
4 2 1
4 3 1
        ===>>
  3  3  3  2  2
  3  3  3  2  2
  3  3  3  2  2
  1  1  1  2  2


Count of minimal permutations: 4
```

```
Enter rectangle height&width:2 3
```

```
Minimal number of squares: 6
1 1 1
1 2 1
1 3 1
2 1 1
2 2 1
2 3 1
        ===>>
  1  1  1
  1  1  1


Count of minimal permutations: 1
```

```
Enter rectangle height&width:6 8
```

```
Minimal number of squares: 6
1 1 4
1 5 4
5 1 2
5 3 2
5 5 2
5 7 2
        ===>>
  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4
  4  4  4  4  4  4  4  4
  2  2  2  2  2  2  2  2
  2  2  2  2  2  2  2  2


Count of minimal permutations: 4
```

Вывод.

В ходе выполнения лабораторной работы был изучен алгоритм поиска с возвратом путем написания программы, решающей задачу квадрарирования прямоугольников.

Приложения А. Исходный код

```
#include <iostream>
#include <vector>
#define DBG
using namespace std;



//Структура вставленного квадрата в виде координаты-размер квадрата
struct InsertedRect{
   int x;
   int y;
   int size;
};

int n, m;
int minCount;              // минимальное число квадратов для покрытия
int colorings = 0;         // количество покрытий минимальным числом квадратов
vector<InsertedRect> minRect;     // минимальное разбиение
vector<vector<int>> resultRect;

//Функция инициализации прямоугольника нулями
void clearRect(vector<vector<int>> &_rect){
   for (int i = 0; i < n; i++) {
      _rect.resize(n);
      for (int j = 0; j < m; j++) {
         _rect[i].push_back(0);
      }
   }
}

void printRect(vector<vector<int>>& _rect){       //Вывод текущего состояния поля в
стандартный поток вывода

   cout << "\t===>>\n";
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         cout.width(3);
         cout << _rect[i][j];
      }
      cout << '\n';
   }
   cout << endl;
}

//Функция восстановления прямоугольника и вектора вставленных квадратов
void recovery(vector<vector<int>>& _rect, vector<InsertedRect> &_res){
   InsertedRect tmp = *(_res.rbegin());                  // сохраняем во временную переменную
последний добавленный квадрат
```

```
   _res.pop_back();                          // удаляем из вектора последний квадрат
   cout << "recovery\n";
   for (int i = tmp.x; i < tmp.x+tmp.size; i++)       // заполняем нулями область, в которой был
размещен последний квадрат
      for (int j = tmp.y; j < tmp.y+tmp.size; j++)
         _rect[i][j] = 0;
}

//Функция проверки на то, можно ли вставить квадрат размера size в точку {x,y}
bool isPushable(vector<vector<int>> &_rect, int x, int y, int size){
   if (n-x < size || m-y < size)             // если квадрат выходит за пределы поля возвращаем
false
      return false;
   for (int i = x; i < x+size; i++)          // если квадрат пересекат другой квадрат возвращаем
false
      for (int j = y; j < y+size; j++){
         if (_rect[i][j])
            return false;
      }
   return true;                          // если можно вставить квадрат возвращаем true
}

//Функция вставки квадрата
void push(vector<vector<int>>& _rect, int x, int y, int size){
   for (int i = x; i < x+size; i++)
      for (int j = y; j < y+size; j++){
         _rect[i][j] = size;
      }
}

//Непосредственно рекурсивный бэктрекинг
void recBack(vector<vector<int>>& _rect, int curSquare, int unPainted, int countOfSquares,
vector<InsertedRect> &_res){
   if ((countOfSquares==minCount-1 && unPainted > curSquare*curSquare))       // если на
текущем шаге дойдем до числа лучшего на данный
      return;                                          // момент разбиения, а поля не будет
закрашено полностью, возвращаемся

   bool pushed = false;
   for (int i = 0; i < n && !pushed; i++){
      for (int j = 0; j < m && !pushed; j++){
         if (_rect[i][j] == 0) {
            if (isPushable(_rect, i, j, curSquare)) {              // если можно вставить в первую
свободную ячейку - вставляем
               pushed = true;
               push(_rect, i, j, curSquare);
               printRect(_rect);
               _res.push_back({i, j, curSquare});
            }
```

```
            else{                                   // иначе возвращаемся
               return;
            }
         }
         else
            j+=_rect[i][j]-1;                       // так как квадраты в матрице заполняются
числом, равным своему размеру,
                                                    // можем проскакивать на это число дальше по
матрице
      }
   }

   if (countOfSquares+1 == minCount) {              // если число квадратов на этом шаге
равно лучшему на данный момент разбиению,
      if (unPainted==curSquare*curSquare)           // то если все закрашено,
увеличиваем счетчик вариантов заполнения минимальным числом квадратов
      {
         colorings++;
         cout << "it's a new minimum\n";
      }
      recovery(_rect, _res);                        // отменяем внесенные изменения и
возвращаемся
      return;
   }
   if (unPainted==curSquare*curSquare && countOfSquares+1 < minCount){ // если все поле
заполнено и текущее число квадратов меньше чем лучшее на данный
      colorings = 1;                                // момент, заменяем лучшее на текущее
      minCount = countOfSquares+1;
      minRect.assign(_res.begin(), _res.end());
      cout << "it's a new minimum\n";
      for (int i = 0; i < n; i++)
         for (int j = 0; j < m; j++)
            resultRect[i][j] = _rect[i][j];

      recovery(_rect, _res);                        // отменяем внесенные изменения и
возвращаемся
      return;
   }

   for (int i = min(min(n,m)-1, max(n,m) - curSquare); i > 0; i--)    // рекурсивно вызываем для
следующих квадратов
      if (i*i <= unPainted)
         recBack(_rect, i, unPainted-curSquare*curSquare, countOfSquares+1, _res);

   recovery(_rect, _res);        // отменяем изменения перед выходом
}

int main() {
   cout << "Enter rectangle height&width: ";
```

```
   cin >> n >> m;                    // считываем размер прямоугольника
   minCount = n*m+1;                 // выставляем недостижимое значение числа квадратов
   int expansionCoeff = 1;           // коэфф расширения для замены квадратов
   vector<vector<int>> rect;
   vector<InsertedRect> result;
   clearRect(resultRect);
   // если квадрат кратен 2, 3, 5 - заменяем его на квадрат размера 2, 3, 5 и записываем
коэффициент расширения для вывода
   if (n==m){
      if (n%2 == 0){
         expansionCoeff = n/2;
         n = 2;
         m = 2;
      }
      else if (n%3 == 0){
         expansionCoeff = n/3;
         n = 3;
         m = 3;
      }
      else if (n%5 == 0){
         expansionCoeff = n/5;
         n = 5;
         m = 5;
      }
   }

   clearRect(rect);       // инициализируем матрицу нулями
   int unPainted = n*m;      // записываем число не закрашенных ячеек

   for (int i = min(n,m)-1; i > 0; i--)   // вызываем бэктрекинг с начальным квадратом размера
от n-1 до 1
      recBack(rect, i, unPainted, result.size(), result);

   cout << "Minimal number of squares: " << minCount << endl;
   for (auto & i : minRect){   // выводим результат с учетом коэффициента расширения
      cout << expansionCoeff*i.x + 1 << ' ' << expansionCoeff*i.y + 1 << ' ' <<
expansionCoeff*i.size << '\n';
   }
   printRect(resultRect);
   cout << "\nCount of minimal permutations: " << colorings << endl;
   return 0;
}
```