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. 9383 ___ Сергиенкова А.А.

Преподаватель ___ Фирсов М.А.

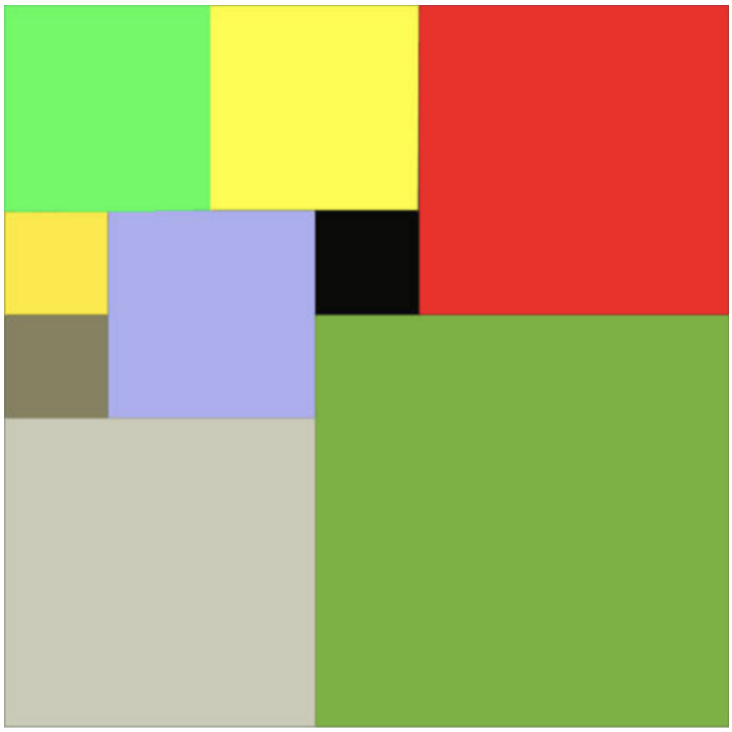
Санкт-Петербург
2021

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов.

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 ($2 \le N \le 40$).

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

Вариант 3р. Рекурсивный бэктрекинг. Исследование кол-ва операций от размера квадрата.

Ход работы:

1. Сначала принимаем данные, выдаём результат и пробуем пропорционально уменьшить размер, для поиска в этом размере.
2. Далее создаётся массив занятых клеток, изначально состоит из 0. И вставляем первые 3 элемента.
3. Поиск первого пустого элемента.
4. Добавление элемента. Если результат лучший – анализируем его.
5. После, алгоритм откатывается назад до тех пор, пока не встретит квадрат размера >1, затем удаляет его и ставит на его место квадрат размер на 1 больше.
6. Конец работы алгоритма наступит тогда, когда удалить квадрат будет невозможно.

Улучшения.

- Размер столешниц, которые кратны 2, 3, 5 число квадратов всегда равно 4, 6 и 8.

- Если размер столешницы число простое, то вся столешница имеет квадраты различных размеров в трех углах: два квадрата (в первом и втором углу) размером n/2, которые касаются сторонами третьего квадрата, в третьем углу находится квадрат размером (n/2) + 1).

Описание функций и структур данных:

- class Position – позиция на столешнице.
- class Tile – элемент столешницы.
- class Resolver – класс для поиска результата.
- void Resolve() – функция поиска результата.
- bool CanAdd () – функция для проверки выхода за границы и для проверки наличия квадрата на конкретном месте.
- bool HasResult() – функция для проверки результата (если истина, то результат получен).
- void Add() – добавляет в раскладку указанный элемент.
- void Remove() – удаляет последний элемент из раскладки.
- bool Scan() – попытка найти результат, начиная с добавления указанного элемента .
- position FindEmpty() – возвращает первое попавшееся пустое место.
- void Resolve2() – возвращает решение для чётного n.
- int main() -- основная функция программы, которая занимается поиском результата.

Тестирование.

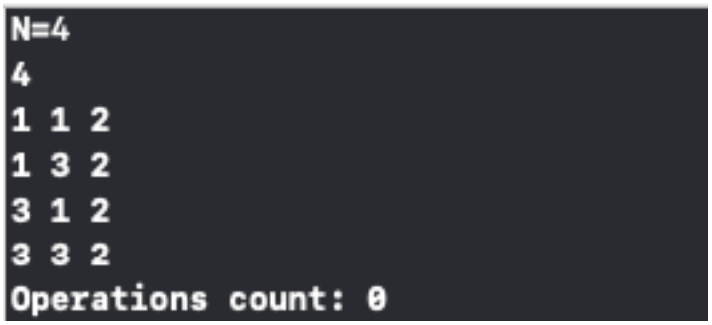


Рисунок 1 - Чётные входные данные.

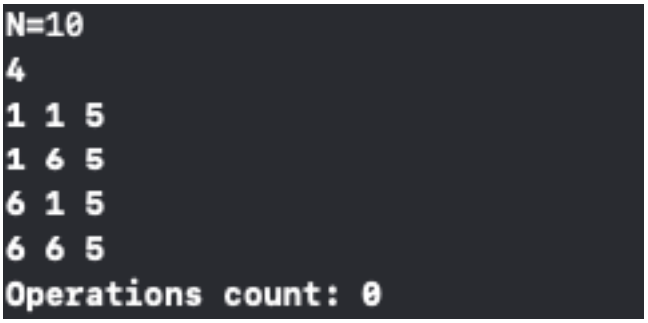


Рисунок 2 - Чётные входные данные 2.

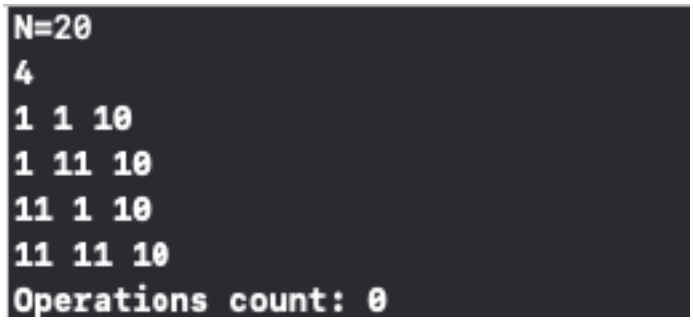


Рисунок 3 - Чётные входные данные 3.

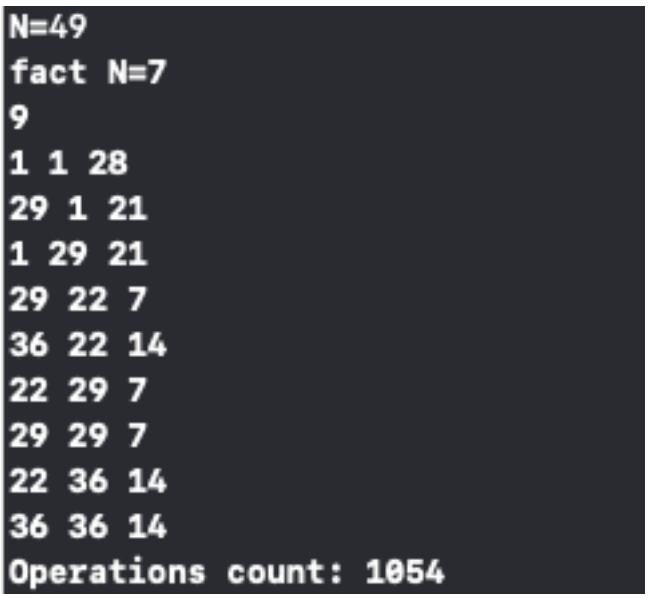


Рисунок 4 - Составные входные данные.

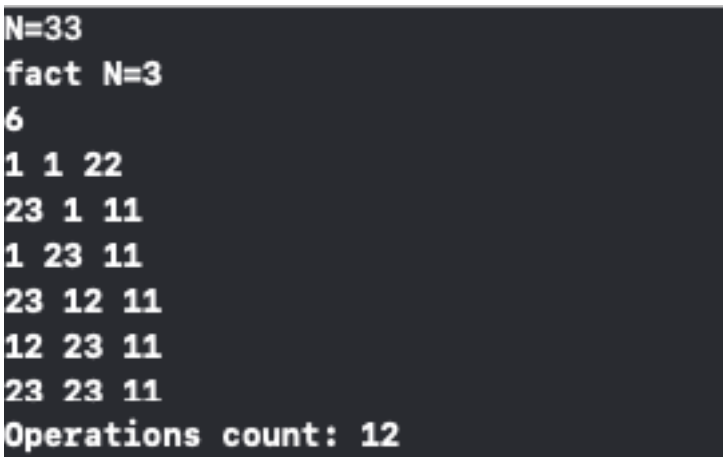


Рисунок 5 - Составные входные данные 2.

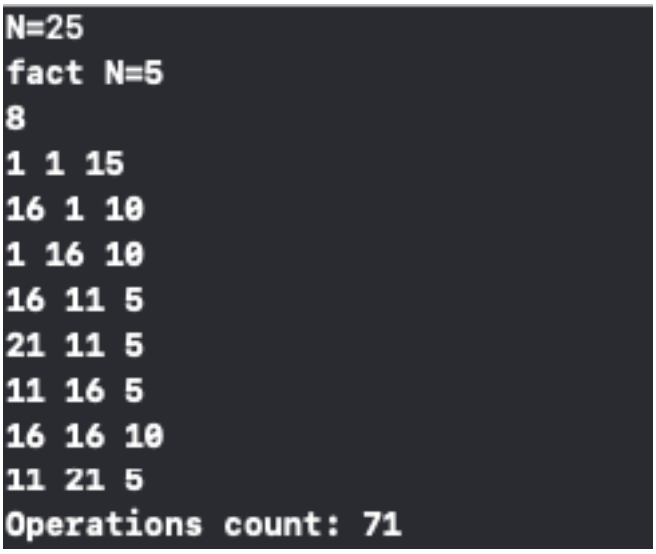


Рисунок 6 - Составные входные данные 3.

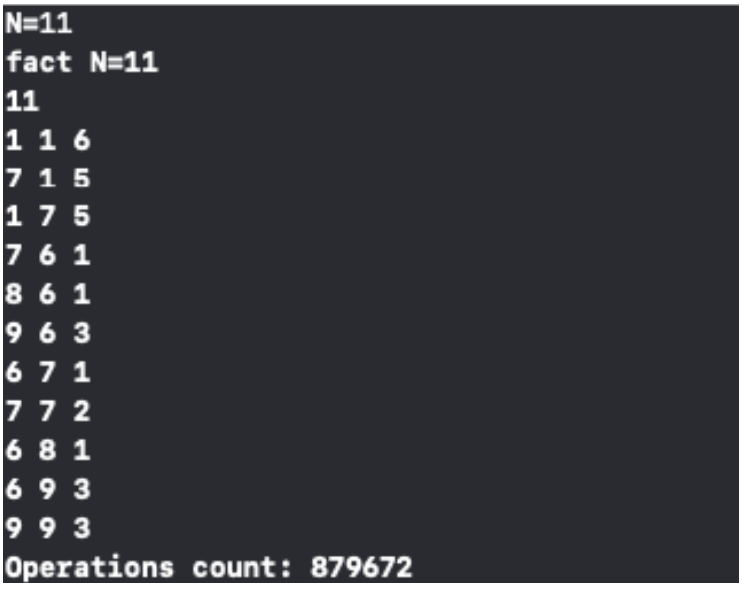


Рисунок 7 - Простые входные данные.

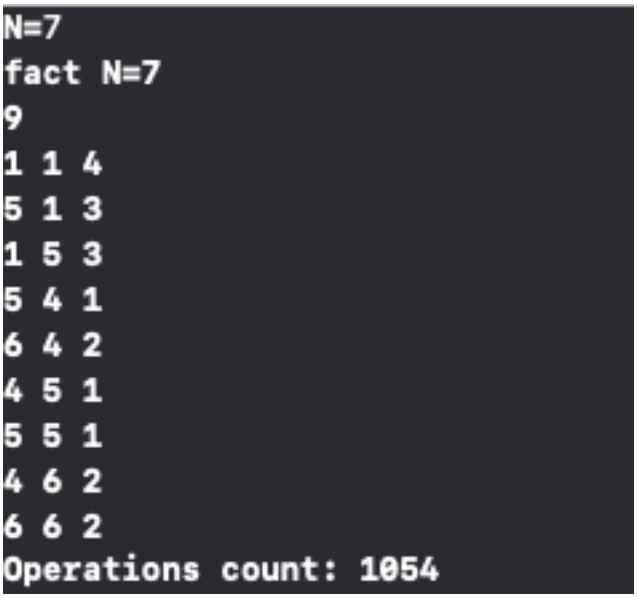


Рисунок 8 - Простые входные данные 2.

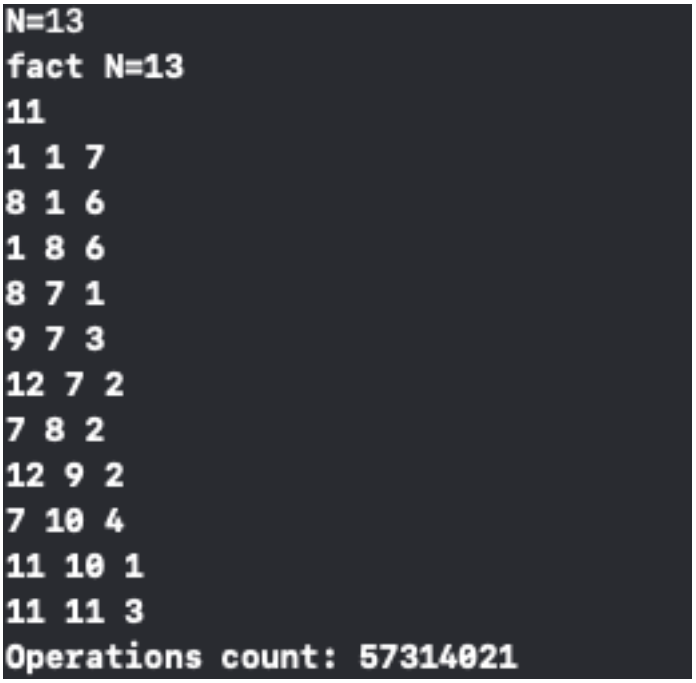


Рисунок 9 - Простые входные данные 3.

Анализ результатов.

Из результатов видно, что при чётном размере столешницы количество операций равно 0, тк там нет подбора. При составном размере столешницы, количество операций довольно маленькое, но при размере столешницы, равному простому числу, количество операций увеличивается с экспоненциальной зависимостью.

Выводы.

Был применён на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <iostream>
#include "Resolver.h"

int main() {
    // ввод исходных данных
    Resolver resolver;
    int n = 40;
    std::cout << "N=";
    std::cin >> n;

    // расчет
    resolver.Resolve(n);

    // вывод результата
    std::cout << resolver << std::endl;

    system("PAUSE");
    return 0;
}
```

Файл Position.cpp:

```
#include "Position.h"

std::ostream& operator<<(std::ostream& os, Position& position)
{
    return os << position.x + 1 << ' ' << position.y + 1;
}

Position::Position(int x, int y)
{
    this->x = x;
    this->y = y;
}
```

Файл Position.h:

```
#pragma once
#include <iostream>

class Position
{
public:
    int x;
    int y;

    Position(int x, int y);

    friend std::ostream& operator<<(std::ostream& os, Position&
position);
};
```

Файл Resolver.cpp:

```
#include "Resolver.h"


std::ostream& operator<<(std::ostream& os, Resolver& resolver)
{
    std::cout << resolver.m_Result.size() << std::endl;
    for (auto i = resolver.m_Result.begin(); i !=
resolver.m_Result.end(); ++i)
     os << *i << std::endl;
    os << "Operations count: " << resolver.m_Operations;
    return os;
}

void Resolver::Resolve(int n)
{
    // сброс количества операций
    m_Operations = 0;

    // принимаем данные
    m_N = n;
    m_Div = 1;

    // если четное то выдаем результат
    if (n % 2 == 0) {
     Resolve2();
     return;
    }
```

```
    // пробуем пропорционально уменьшить размер, для поиска в этом
размере
    for (int i = 2; i < m_N; ++i) {
     if (m_N % i == 0) { // если можно
          m_Div = m_N / i;
          m_N = i;
          break;
     }
    }
    std::cout << "fact N=" << m_N << std::endl;// с каким размером
идет фактический расчет

    // очистка
    m_Result.clear();
    m_Tiles.clear();

    // создаем массив занятых клеток
    m_Data = new int*[m_N];
    for (int i = 0; i < m_N; ++i) m_Data[i] = new int[m_N];

    // зануляем массив занятых клеток
    for (int i = 0; i < m_N; ++i)
     for (int j = 0; j < m_N; ++j)
          m_Data[i][j] = 0;

    // вставка первых 3 элементов
    Add(Tile(0, 0, m_N / 2 + 1));
    Add(Tile(m_N / 2 + 1, 0, m_N / 2));
    Add(Tile(0, m_N / 2 + 1, m_N / 2));

    // ищем первый пустой элемент
    auto pos = FindEmpty();

    // запуск перебора всех раскладок
    Scan(Tile(pos.x, pos.y, 1));

    // удаляем массив занятых клеток
    for (int i = 0; i < m_N; ++i) delete[] m_Data[i];

    // восстанавливаем размер
    m_N *= m_Div;
    for (auto i = m_Result.begin(); i != m_Result.end(); ++i) {
     (*i).Position.x *= m_Div;
     (*i).Position.y *= m_Div;
     (*i).Size *= m_Div;
    }

    delete[] m_Data;
}
bool Resolver::CanAdd(Tile tile)
{
    // если тайл выходит за рамки то нельзя
```

```
    if (tile.Position.x + tile.Size > m_N) return false;
    if (tile.Position.y + tile.Size > m_N) return false;

    // если на месте тайла уже есть тайл то нельзя
    for (int y = tile.Position.y; y < tile.Position.y + tile.Size;
++y)
     for (int x = tile.Position.x; x < tile.Position.x + tile.Size;
++x)
          if(m_Data[x][y]) return false;

    // если все ок
    return true;
}
bool Resolver::HasResult()              // если истина, то получен
результат
{
    // если где-то есть пустота то результата нет
    for (int y = 0; y < m_N; ++y)
     for (int x = 0; x < m_N; ++x)
          if (m_Data[x][y] == 0) return false;

    // если нигде нет то ок
    return true;
}
Position Resolver::FindEmpty()          // возвращает первое
попавшееся пустое место
{
    // если гдето есть пустота то результата нет
    for (int y = m_N / 2; y < m_N; ++y) {
     if (y >= m_N / 2 + 1) {
          for (int x = m_N / 2; x < m_N; ++x) {
               ++m_Operations;
               if (m_Data[x][y] == 0) return Position(x, y);
          }
     }
     else {
          for (int x = m_N / 2 + 1; x < m_N; ++x) {
               ++m_Operations;
               if (m_Data[x][y] == 0) return Position(x, y);
          }
     }
    }

    // если гдето есть пустота то результата нет
    /*for (int y = 0; y < m_N; ++y) {
     for (int x = 0; x < m_N; ++x) {
          ++m_Operations;
          if (m_Data[x][y] == 0) return Position(x, y);
     }
    }*/

    // если не найдено
```

```
    throw "has no empty position";
}
void Resolver::Remove()                 // удаляет последний тайл из
раскладки
{
    // берем последний тайл
    auto tile = m_Tiles.back();
    // удаляем из списка
    m_Tiles.pop_back();
    // пишем данные в массив
    for (int y = tile.Position.y; y < tile.Position.y + tile.Size;
++y)
     for (int x = tile.Position.x; x < tile.Position.x + tile.Size;
++x)
          m_Data[x][y] = 0;
}
void Resolver::Add(Tile tile)
{
    // увеличиваем количество операций
    ++m_Operations;
    // вставка в список
    m_Tiles.push_back(tile);
    // пишем данные в массив
    for (int y = tile.Position.y; y < tile.Position.y + tile.Size;
++y)
     for (int x = tile.Position.x; x < tile.Position.x + tile.Size;
++x)
          m_Data[x][y] = 1;
}
bool Resolver::Scan(Tile tile)          // попытка найти результат,
начиная с добавления указанного тайла
{
    // ограничитель
    if (tile.Size >= m_N) return false;
    if (!CanAdd(tile)) return false;

    // добавляем тайл
    Add(tile);

    // если есть результат - анализируем его
    if (HasResult()) {
     // если рузультат лучший - запоминаем его
     if (m_Tiles.size() < m_Result.size() || m_Result.size() == 0)
          m_Result = m_Tiles;
     // откат назад
     /*Remove();
     // скан с большим тайлом
     ++tile.Size;
     Scan(tile);
     // результат был - далее не сканируем (истина тк просканировано)
     return true;*/
    }
```

```
    else {
     // если результата нет – сканируем все возможные квадраты на
первом попавшемся месте
     // берем первое попавшееся пустое место
     auto position = FindEmpty();
     // начинаем скан с этой свободной позиции
     tile = Tile(position.x, position.y, 1);
     while (Scan(tile)) ++tile.Size;
    }

    // откат назад
    Remove();

    // вывод истины, тк просканировано
    return true;
}
void Resolver::Resolve2()              // возвращает решение для
четного n
{
    m_Result.push_back(Tile(0, 0, m_N / 2));
    m_Result.push_back(Tile(0, m_N / 2, m_N / 2));
    m_Result.push_back(Tile(m_N / 2, 0, m_N / 2));
    m_Result.push_back(Tile(m_N / 2, m_N / 2, m_N / 2));
}
```

Файл Resolver.h:

```
#pragma once
#include <iostream>
#include <list>
#include "Tile.h"

class Resolver // решает задачу
{
    std::list<Tile> m_Result; // результат работы
    std::list<Tile> m_Tiles;  // список для перебора раскладок
    int m_N;                        // размер стола
    int m_Div;                      // во сколько раз уменьшились при
вычислениях результата
    int** m_Data;                   // двумерный массив занятых клеток
    unsigned int m_Operations;      // количество операций при
вичислении

    bool CanAdd(Tile tile);         // возвращает истину, если в
текущую раскладку можно добавить указанный тайл
    void Add(Tile tile);       // добавляет в раскладку указанный тайл
    void Remove();                  // удаляет последний тайл из
раскладки
```

```
    bool Scan(Tile tile);          // попытка найти результат,
начиная с добавления указанного тайла. Истина, если с этим тайлом
скан возможен
    bool HasResult();              // если истина, то получен
результат
    Position FindEmpty();          // возвращает первое попавшееся
пустое место
    void Resolve2();               // возвращает решение для четного
n
public:

    void Resolve(int n);
    friend std::ostream& operator<<(std::ostream& os, Resolver&
resolver);
};
```

Файл Tile.cpp:

```
#include "Tile.h"


std::ostream& operator<<(std::ostream& os, Tile& tile)
{
    return os << tile.Position << ' ' << tile.Size;
}

Tile::Tile(int x, int y, int size) :
    Position(x, y)
{
    this->Size = size;
}
```

Файл Tile.h:

```
#pragma once
#include <iostream>
#include "Position.h"

class Tile
{
public:
    Position Position;
    int Size;

    Tile(int x, int y, int size);

    friend std::ostream& operator<<(std::ostream& os, Tile& tile);
```

```
};
```