HSE-A&DT-LAB2

Постановка задачи

Даны прямоугольники на плоскости с углами в целочисленных координатах **([1..10^9^],[1..10^9^])**. Требуется как можно быстрее выдавать ответ на вопрос «Скольким прямоугольникам принадлежит точка **(x,y)** ?» Подготовка данных должна занимать мало времени.

Цели

- Реализовать **три разных решения** задачи;
- Выяснить при каком объеме начальных данных и точек какой алгоритм эффективнее.

Алгоритм №1 | Простой перебор

Первый алгоритм представляет из себя наиболее тривиальное решение, работающее без подготовки данных за время **O(N)**.

```
std::vector<int> enumerationSolution(const std::vector<Rectangle>& rectangles, std::vector<Point2D>& targets)
{
    std::vector<int> answers(targets.size(), 0);
    for (size_t i = 0; i < targets.size(); ++i)
    {
        int result = 0;
        for (const auto r : rectangles)
        {
            result += (targets[i].getX() >= r.getLeftLowerAngle().getX() &&   //
                       targets[i].getX() <= r.getRightUpperAngle().getX() &&  //
                       targets[i].getY() >= r.getLeftLowerAngle().getY() &&   //
                       targets[i].getY() <= r.getRightUpperAngle().getY());
        }
        answers[i] = result;
    }
    return answers;
}
```

Алгоритм №2 | Карта сжатых координат

Идея второго алгоритма заключается в сжатии изначальных координат и построении на них карты, обеспечивающий доступ к элементу за **O(logN)**. Для работы этого алгоритма необходима предварительная подготовка данных, сложность которой составляет **O(N^3^)**.

Реализация сжатия координат

```
std::pair<std::vector<int>, std::vector<int>> compressCoordinatesCM(const std::vector<Rectangle>& rectangles)
{
    std::vector<int> uniqueCoordsX, uniqueCoordsY;
    for (auto& r : rectangles)
    {
        uniqueCoordsX.push_back(r.getLeftLowerAngle().getX());
        uniqueCoordsX.push_back(r.getRightUpperAngle().getX());
        uniqueCoordsX.push_back(r.getRightUpperAngle().getX() + 1);
        uniqueCoordsY.push_back(r.getLeftLowerAngle().getY());
        uniqueCoordsY.push_back(r.getRightUpperAngle().getY());
        uniqueCoordsY.push_back(r.getRightUpperAngle().getY() + 1);
    }
    sort(uniqueCoordsX.begin(), uniqueCoordsX.end());
    sort(uniqueCoordsY.begin(), uniqueCoordsY.end());
    uniqueCoordsX.erase(std::unique(uniqueCoordsX.begin(), uniqueCoordsX.end()), uniqueCoordsX.end());
    uniqueCoordsY.erase(std::unique(uniqueCoordsY.begin(), uniqueCoordsY.end()), uniqueCoordsY.end());
    return std::pair<std::vector<int>, std::vector<int>>(uniqueCoordsX, uniqueCoordsY);
}
```

Реализация алгоритма

```
std::vector<int> compressedMapSolution(const std::vector<Rectangle> rectangles, std::vector<Point2D> targets)
{
    auto [uniqueCoordsX, uniqueCoordsY] = compressCoordinatesCM(rectangles);
    std::vector<std::vector<int>> map(uniqueCoordsX.size());
    for (std::size_t i = 0; i < uniqueCoordsX.size(); ++i)
    {
        map[i].resize(uniqueCoordsY.size());
    }
    for (const auto& r : rectangles)
    {
        Point2D compressedRightUpper(
            findPosition(uniqueCoordsX, r.getRightUpperAngle().getX()), findPosition(uniqueCoordsY, r.getRightUpperAngle().getY()));
        Point2D compressedLeftDown(
            findPosition(uniqueCoordsX, r.getLeftLowerAngle().getX()), findPosition(uniqueCoordsY, r.getLeftLowerAngle().getY()));
        for (long xIdx = compressedLeftDown.getX(); xIdx < compressedRightUpper.getX() + 1; xIdx++)
        {
            for (long yIdx = compressedLeftDown.getY(); yIdx < compressedRightUpper.getY() + 1; yIdx++)
            {
                map[xIdx][yIdx]++;
            }
        }
    }
    std::vector<int> answers;
    answers.reserve(targets.size());
    for (const auto& p : targets)
    {
        size_t compressedX = findPosition(uniqueCoordsX, p.getX());
        size_t compressedY = findPosition(uniqueCoordsY, p.getY());
        answers.emplace_back(map[compressedX][compressedY]);
    }
    return answers;
}
```

Алгоритм №3 | Персистентное дерево отрезков

Третий алгоритм развивает идею второго и также использует сжатие координат, однако карта в нём заменена на персистентное дерево отрезков, использование которого позволяет сократить время, необходимое для подготовки данных до **O(N*logN)**. Время, необходимое для выдачи ответа алгоритмом без учета подготовки данных осталось на уровне **O(logN)**.

Реализация сжатия координат

```
std::pair<std::vector<int>, std::vector<int>> compressCoordinatesPST(const std::vector<Rectangle>& rectangles)
{
    std::vector<int> uniqueCoordsX, uniqueCoordsY;
    for (auto& r : rectangles)
    {
        uniqueCoordsX.push_back(r.getLeftLowerAngle().getX());
        uniqueCoordsX.push_back(r.getRightUpperAngle().getX() + 1);
        uniqueCoordsY.push_back(r.getLeftLowerAngle().getY());
        uniqueCoordsY.push_back(r.getRightUpperAngle().getY() + 1);
    }
    sort(uniqueCoordsX.begin(), uniqueCoordsX.end());
    sort(uniqueCoordsY.begin(), uniqueCoordsY.end());
    uniqueCoordsX.erase(std::unique(uniqueCoordsX.begin(), uniqueCoordsX.end()), uniqueCoordsX.end());
    uniqueCoordsY.erase(std::unique(uniqueCoordsY.begin(), uniqueCoordsY.end()), uniqueCoordsY.end());
    return std::pair<std::vector<int>, std::vector<int>>(uniqueCoordsX, uniqueCoordsY);
}
```

Реализация Persistent Segment Tree

```
struct Node
{
    Node() {}
    Node(const std::shared_ptr<Node> other) : value(other->value), left(other->left), right(other->right) {}
    std::shared_ptr<Node> left = nullptr;
    std::shared_ptr<Node> right = nullptr;
    int value = 0;
};

std::shared_ptr<Node> newPersistentConditionOfTree(std::shared_ptr<Node> node, int l, int r, int value, int range_left, int range_right)
{
    if (std::max(l, range_left) <= std::min(r, range_right))
    {
        std::shared_ptr<Node> root = node ? std::make_shared<Node>(node) : std::make_shared<Node>();
        if (range_left >= l && range_right <= r)
        {
            root->value += value;
            return root;
        }
        int middle = (range_left + range_right) / 2;
        root->left = newPersistentConditionOfTree(node ? node->left : nullptr, l, r, value, range_left, middle);
        root->right = newPersistentConditionOfTree(node ? node->right : nullptr, l, r, value, middle + 1, range_right);
        return root;
    }
    return node;
}
```

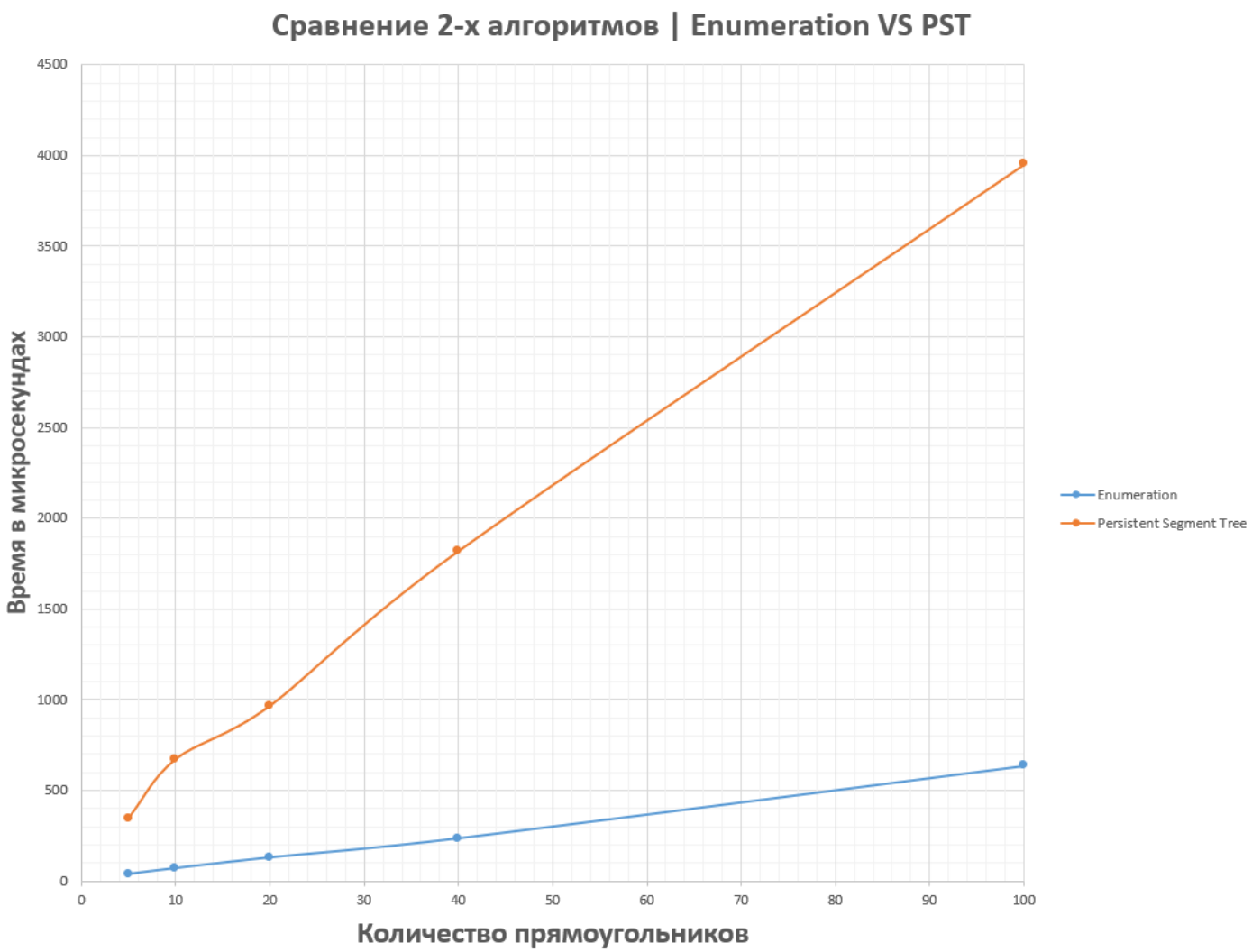
Реализация алгоритма

```
std::vector<int> persistentSegmentTreeSolution(const std::vector<Rectangle>& rectangles, const std::vector<Point2D>& targets)
{
    auto [uniqueCoordsX, uniqueCoordsY] = compressCoordinatesPST(rectangles);
    std::vector<std::pair<int, Rectangle>> coordinates;
    for (auto& r : rectangles)
    {
        coordinates.push_back(std::make_pair(findPositionPST(uniqueCoordsX, r.getLeftLowerAngle().getX()), r));
        coordinates.push_back(std::make_pair(findPositionPST(uniqueCoordsX, r.getRightUpperAngle().getX() + 1), r));
    }
    sort(coordinates.begin(), coordinates.end(),
        [](std::pair<int, Rectangle> left, std::pair<int, Rectangle> right) { return left.first < right.first; });
    std::vector<std::shared_ptr<Node>> trees(rectangles.size() * 2);
    auto root = std::make_shared<Node>();
    int current_coord = 0;
    for (const auto x : coordinates)
    {
        auto lower = findPositionPST(uniqueCoordsY, x.second.getLeftLowerAngle().getY());
        auto upper = findPositionPST(uniqueCoordsY, x.second.getRightUpperAngle().getY());
        if (x.first != current_coord)
        {
            trees[current_coord] = root;
            current_coord = x.first;
        }
        root = newPersistentConditionOfTree(root,                                    //
            lower,                                                                   //
            upper,                                                                   //
            uniqueCoordsX[x.first] == x.second.getLeftLowerAngle().getX() ? 1 : -1,  //
            0,                                                                       //
            uniqueCoordsY.size() - 1);
    }
    std::vector<int> answers;
    for (auto point : targets)
    {
        if (point.getX() < uniqueCoordsX[0] || point.getX() > uniqueCoordsX[uniqueCoordsX.size() - 1] || point.getY() < uniqueCoordsY[0] ||
            point.getY() > uniqueCoordsY[uniqueCoordsY.size() - 1])
        {
            answers.push_back(0);
            continue;
        }
        auto x = findPositionPST(uniqueCoordsX, point.getX());
        auto y = findPositionPST(uniqueCoordsY, point.getY());
        answers.push_back(findRectanglesCount(trees[x], y, 0, uniqueCoordsY.size() - 1));
    }
    return std::vector<int>(answers);
}
```

Вывод | Графики, сравнения и умозаключения

Алгоритм №1

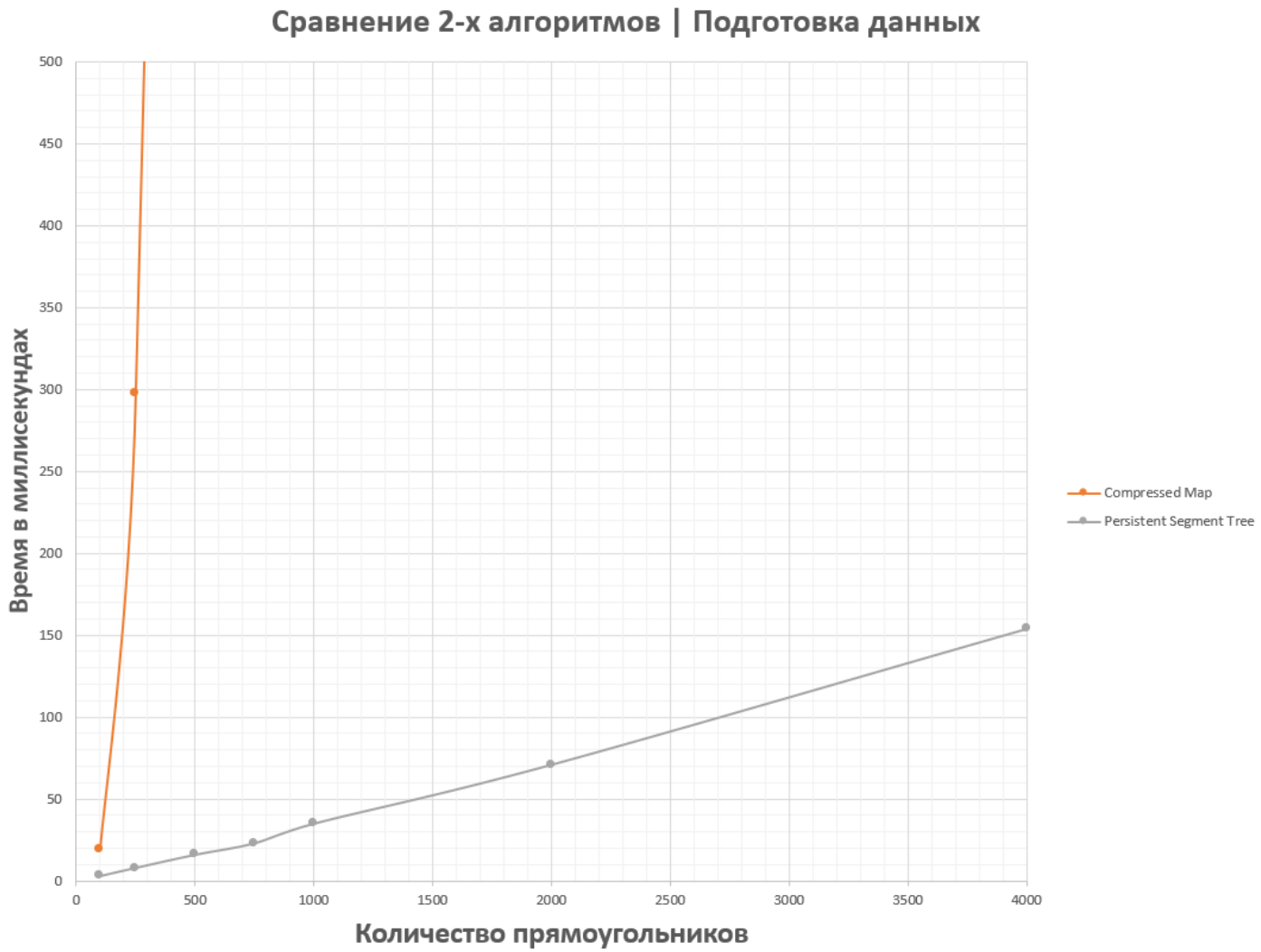
Алгоритм №1 является оптимальным выбором при небольших количествах данных (как прямоугольников, так и точек). Его преимущество наиболее заметно на следующем графике:



При малом количестве точек **(~100)** и прямоугольников **(5-100)** он работает значительно быстрее Алгоритма №3, что делает его самым эффективным решениям для похожего набора входных данных.

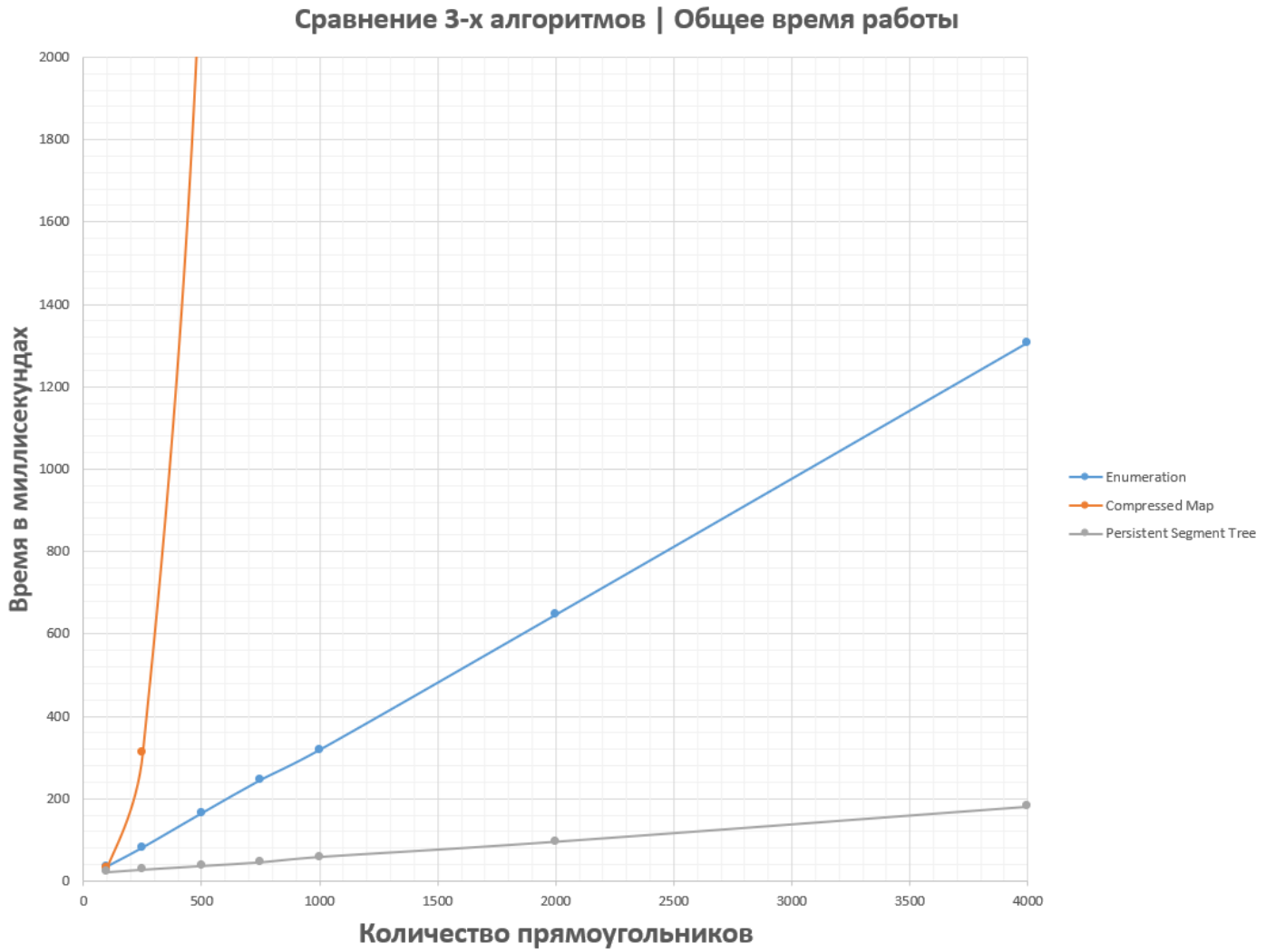
Алгоритм №2

Алгоритм №2 является наиболее медленным из трех рассматриваемых алгоритмов. Из-за колоссального количества времени, требуемого для подготовки данных, целесообразность использования этого алгоритма понижается. Использование рекомендуется, если необходимо единожды построить карту при небольших количествах прямоугольников, а позже регулярно искать количество прямоугольников в заданной точке.



Алгоритм №3

Алгоритм №3, использующий сжатие координат и построение персистентного дерева отрезков является наиболее оптимальным выбором, когда количество входных данных велико или вовсе неизвестно. При средних и больших количествах прямоугольников и точек этот алгоритм значительно обходит в скорости рассмотренные аналоги, что легко заметить на следующем графике:



Итог

Алгоритм №1 целесообразно использовать при небольших количествах данных.

Алгоритм №2 целесообразно использовать при отсутствии необходимости перестройки карты и большом количестве запросов.

Алгоритм №3 является оптимальным выбором, потенциал которого раскрывается при больших количествах данных.