|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
| МИНОБРНАУКИ РОССИИ | | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** | | | |
| **Институт искусственного интеллекта** | | |  |
| **Кафедра высшей математики** | | |  |

|  |  |
| --- | --- |
| **КУРСОВАЯ РАБОТА** | |
| **по дисциплине** | |
| **«**Алгоритмы и теория сложности**»** | |
| **Тема курсовой работы**  **«Алгоритм Грэхэма»** | |
| Студент группы КМБО-07-22 | *Баттур Ц.* |
| Руководитель курсовой работы | *Драгилева И.П.* |
|  |  |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работа представлена к защите | «\_\_»\_\_\_\_\_\_\_2024 г. | *(подпись студента)* |
|  |  |  |
| «Допущен(ы) к защите» | «\_\_»\_\_\_\_\_\_\_2024г. | *(подпись руководителя)* |

Москва – 2024

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** | | | | **Институт искусственного интеллекта** | | | | **Кафедра высшей математики** | | | | | | |  |
|  | | **Утверждаю** | | |
|  | | Заведующий  кафедрой\_\_\_\_\_\_\_\_\_\_\_\_\_Шатина А.В. | | |
|  | | «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_2024г. | | |
| **ЗАДАНИЕ** | | | | |
| **на выполнение курсовой работы** | | | | |
| **по** **дисциплине** «Алгоритмы и теория сложности» | | | | |
|  | | | | |
| Студент Баттур Ц. Группа *КМБО-07-22* | | | | |
|  | | | | |
| 1. **Тема: «Алгоритм Грэхэма»** | | | | |
| **2. Исходные данные:** тестовые примеры и наборы данных для испытаний | | | | |
| **3**. **Перечень вопросов, подлежащих обработке, и обязательного графического материала:**  1) Алгоритм(ы) решения задачи на языке высокого уровня.  2) Получение и сравнение количественных результатов при различных исходных данных.  3) Асимптотические оценки временной сложности. | | | | |
|  | | | | |
| **4. Срок представления к защите курсовой работы:** **до** «23 » декабря 2024 г. | | | | |
|  | | | | |
| Задание на курсовую  работу выдал | « 1» октября 2024г . | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | (Драгилева) | |
| Задание на курсовую  работу получил | «\_\_\_»\_\_\_\_\_\_2024г. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | (Баттур) | |

**Оглавление**

[**Введение** 4](#_Toc182784378)

[**Постановка Задачи** 5](#_Toc182784379)

[**Теоретическая Часть** 6](#_Toc182784380)

[**Описание Алгоритма** 7](#_Toc182784381)

[**Доказательства Корректности** 8](#_Toc182784382)

[**Асимптотическая Оценка Временной Сложности Алгоритма** 10](#_Toc182784383)

[**Результаты Запуска Программы** 11](#_Toc182784384)

[**Заключение** 17](#_Toc182784385)

[**Список Литературы** 18](#_Toc182784386)

[**Приложения** 19](#_Toc182784387)

# **Введение**

Выпуклой оболочкой множества *X*называется наименьшее выпуклое множество, содержащее *X*. Наименьшее множество здесь означает наименьший элемент по отношению к вложению множеств, то есть такое выпуклое множество, содержащее данную фигуру, что оно содержится в любом другом выпуклом множестве, содержащем данную фигуру. Это концепция в вычислетельной геометрии, которая находит применение в различных областях, как компьтерная графика, обработка изображений и обнаружение столкновений.

Выпуклый многоугольник – это многоугольник, у которого все внутренние углы меньше 180 градусов. Выпуклую оболочку можно построить для любого набора точек, независимо от их расположения.

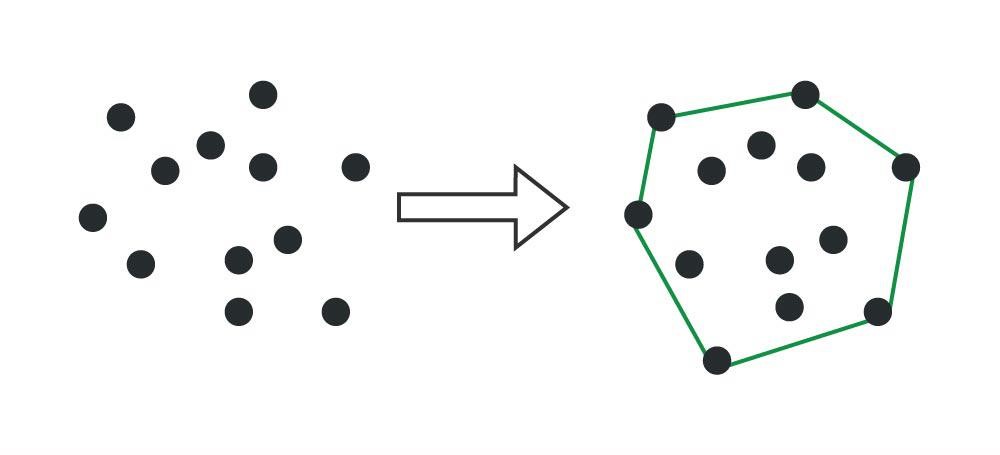


Рисунок 1. Выпуклая оболочка

# **Постановка Задачи**

* 1. Построение минимальной выпуклой оболочки на плоскости
     1. Тестовый пример: 8 точек с координатами (1, 1), (2, 5), (1, 9), (4, 3), (6, 4), (5, 7), (8, 2), (8, 9)
     2. Набор данных испытания: равномерно выбрать 1000 точек из квадрате [-100, 100] x [-100; 100]
  2. Сравнить временные затраты для с затратами задач, использующих рекурсивный алгоритм и алгоритм Джарвиса
  3. Привести и обосновать асимптотическую оценку временной сложности алгоритма.

# **Теоретическая Часть**

Алгоритм Грэхэма – это простой и эффективный алгоритм вычиления выпуклой оболочки множества точек на плоскости с временной сложностью. Он работает путем итеративного добавления точек к выпуклой оболочке до тех пор, пока не все точки не добавлены.

Основные шаги добавления:

1. Поиск опорной точки: Выбтрается токи с наименьшей координатой y. Эта точка всегда находится на выкуклой оболочке.
2. Сортировка точки по полярному углу: Все остальные точки сортируются относительно опорной точки по их полярному углу.
3. Построение выпуклой оболочки: Алгоритм итеративно добавляет точки к выпуклой оболочки. На каждом шаге алгоритм проверяет, образуют ли последние две точки, добавленные к выпуклой оболочки, правый поворот. Если это так, то последняя точка удаляется из выпуклой оболочки. В противном случае следующая точка в отсортированном списке добавляется к выпуклой оболочки.
4. После обработки всех точек в стеке остаются вершины выпуклой оболочки в порядке обхода.

Псевдокод алгоритма Грэхэма для построения выпуклой оболочки:

GrahamScan(S)

1. Найти точку P0 с наименьшей y-координатой (при равенстве — с наименьшей x-координатой)

2. Отсортировать остальные точки в S по возрастанию полярного угла относительно P0

3. Создать пустой стек Stack

4. Поместить P0, P1, P2 в Stack

5. Для i от 3 до n:

a. Пока размер Stack > 1 и поворот (NextToTop(Stack), Top(Stack), Pi) не является левым:

i. Удалить Top(Stack)

b. Добавить Pi в Stack

6. Возвратить Stack (вершины выпуклой оболочки в порядке обхода)

# **Описание Алгоритма**

|  |  |
| --- | --- |
| Алгоритм | Время |
| Алгоритм Джарвиса |  |
| Рекурсивный алгоритм |  |
| Алгоритм Грэхема |  |

Таблица 1: Сравнение времени выполнения

Алгоритм имеет временную сложность , где — это количество точек во входном множестве S. Алгоритм включает предварительную сортировку точек по углу относительно начальной точки, что занимает . После сортировки точки обрабатываются линейно для построения выпуклой оболочки, что занимает . Таким образом, общее время выполнения определяется доминирующим фактором .

Алгоритм Грэхема не зависит от количества точек на выпуклой оболочке и всегда выполняется за , вне зависимости от формы входного множества. Поэтому он является предпочтительным выбором для случаев, когда число точек на оболочке () может быть больше , или когда нужно гарантировать производительность независимо от выходных данных.

# **Доказательства Корректности**

Для доказательства корректности алгоритма Грэхэма необходимо показать, что он действительно строит выпуклую оболочку множества точек на плоскости.

1. Лемма о сортировке:

Формулировка: Сортировка всех точек относительно опорной точки P0​ по полярному углу гарантирует, что обход точек в отсортированном порядке соответствует обходу внешней границы множества точек.

Доказательство:

* 1. **Опорная точка** P0​**:** Выбирается точка с наименьшей координатой y (и при равенстве — с наименьшей координатой x). Эта точка гарантированно принадлежит выпуклой оболочке.
  2. Полярный угол: Для каждой точки Pi (кроме P0) определяется угол между положительным направлением оси x и вектором P0Pi​.
  3. Сортировка: Все точки сортируются в порядке возрастания полярного угла относительно P0​. При равенстве углов точки с большим расстоянием от P0 располагаются дальше.
  4. Аргументация: Если бы точки не были отсортированы по полярному углу, то возможны ситуации, когда обход выпуклой оболочки включал бы внутренние точки или пропускал бы внешние. Сортировка обеспечивает последовательный обход точек вокруг P0, что предотвращает такие ошибки.

Таким образом, сортировка по полярному углу обеспечивает правильный порядок обхода точек, соответствующий внешней границе множества.

1. Монотонность стека:

Формулировка: Во время выполнения алгоритма стек содержит последовательность точек, образующих монотонный по углу порядок, что соответствует части выпуклой оболочки.

Доказательство:

* 1. Инвариант: После добавления каждой новой точки стек содержит вершины текущей выпуклой оболочки в порядке обхода.
  2. Итеративное добавление: При добавлении новой точки Pi​ проверяется ориентация последних двух точек в стеке с Pi. Если ориентация образует правый поворот, предыдущая точка удаляется, так как она не принадлежит выпуклой оболочке.
  3. Монотонность: Удаление точек при правом повороте гарантирует, что последовательность точек в стеке остается монотонной по полярному углу, предотвращая формирование внутренних (невыпуклых) углов.
  4. Завершение: В конце обработки всех точек стек содержит только те точки, которые необходимы для формирования выпуклой оболочки, и они расположены в правильном порядке.

Таким образом, инвариант монотонности стека сохраняется на каждом шаге алгоритма, обеспечивая корректное построение выпуклой оболочки.

1. Удаление внутренних точек:

Формулировка: При обнаружении правого поворота внутренние точки удаляются из стека, гарантируя, что оставшиеся точки находятся на выпуклой оболочке.

Доказательство:

* 1. Правый поворот: Если добавление точки Pi вместе с двумя предыдущими точками Pi−1​ и Pi−2​ образует правый поворот, это означает, что Pi−1​ находится внутри предполагаемой оболочки и не может быть вершиной выпуклой оболочки.
  2. Удаление: Удаление Pi−1​ из стека устраняет внутреннюю точку, позволяя строить оболочку, обходя только внешние точки.
  3. Минимальность: Продолжая этот процесс до тех пор, пока не будет достигнуто левое или коллинеарное направление, алгоритм гарантирует, что только минимальный набор точек, образующих выпуклую оболочку, остается в стеке.

Таким образом, удаление внутренних точек при правом повороте обеспечивает, что финальный стек содержит только вершины выпуклой оболочки, исключая все внутренние точки.

# **Асимптотическая Оценка Временной Сложности Алгоритма**

1. Поиск опорной точки: . Для нахождения такой точки необходимо пройти один раз по всем точкам.
2. Сортировка точек: . После выбора опорной точки все остальные точки сортируются по полярному углу относительно неё.
3. Обход и построение стека: . Стек используется для хранения вершин выпуклой оболочки, и каждая точка добавляется или удаляется из стека не более одного раза

Общая временная сложность складывается из указанных этапов и составляет , где наиболее затратным по времени является этап сортировки.

# **Результаты Запуска Программы**

Три алгоритма были протестированы для построения выпуклой оболочки: Алгоритм Жарвиса, Алгоритм Грэхэма и Рекурсивный алгоритм для построения выпуклой оболочки.

Первое тестирование проводилось на данных из 8 точек: (1, 1), (2, 5), (1 ,9), (4, 3), (6, 4), (5,7), (8, 2), (8, 9).

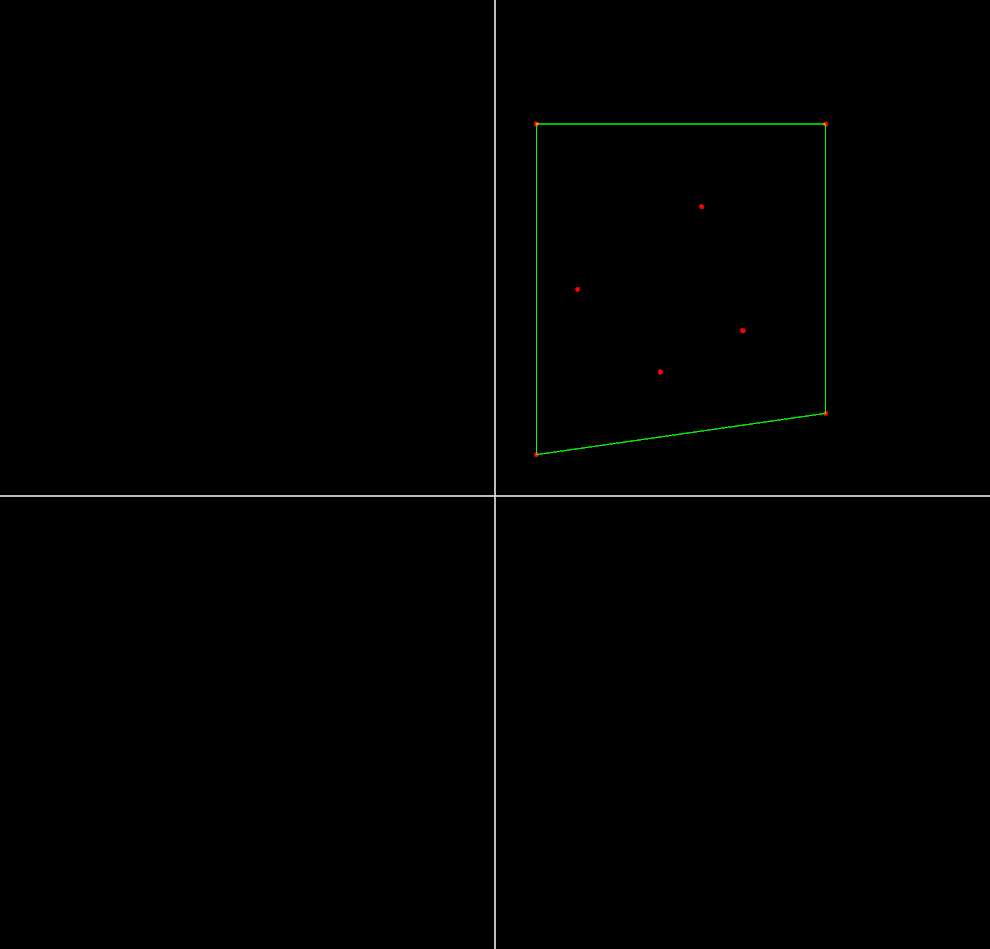


Рисунок 2. Выпуклая оболочка



Рисунок 3. Результаты запуска алгоритмов из примеров точек

Тестирование проводилось на том же данных из 1000 точек, равномерно выбранных из квадрата , для обеспечения корректного сравнения.

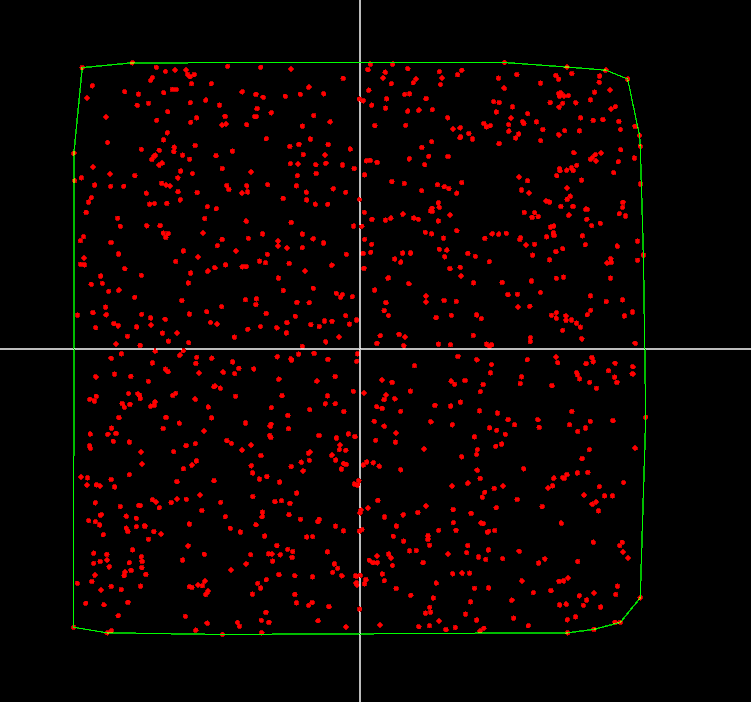


Рисунок 3. Выпуклая оболочка из 1000 точек

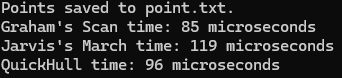


Рисунок 4. Первый результат для 1000 точек

* Алгоритм Жарвиса: 119 микросекундов
* Рекурсивный Алгоритм: 96 микросекундов
* Алгоритм Грэхэма: 85 микросекундов

Анализ результатов:

* Алгоритм Грэхэма показал наилучшее время выполнения – 85 микросекунд. Его временная сложность — O(nlogn), что обеспечивает стабильную производительность
* Рекурсивный алгоритм (QuickHull) занял 96 микросекунд. Он продемонстрировал хорошую производительность на случайных наборах точек благодаря разделяй и властвуй подходу.
* Алгоритм Жарвиса занял 119 микросекунд, что является наибольшим временем среди тестируемых алгоритмов. Его временная сложность — O(nh), где h — количество вершин выпуклой оболочки.



Рисунок 5. Второй результат для 1000 точек



Рисунок 6. Третий результат для 1000 точек

В зависимости от расположения точек результаты работы различных алгоритмов различаются. Однако, на основе предоставленных данных:

Алгоритм Грэхэма consistently показывает наилучшее время выполнения, варьируясь в диапазоне от 85 до 88 микросекунд. Его временная сложность O(nlogn) обеспечивает высокую производительность при тестируемых наборах точек.

QuickHull (рекурсивный алгоритм) показывает результаты от 92 до 96 микросекунд. Этот алгоритм демонстрирует стабильное время выполнения благодаря подходу «разделяй и властвуй» и является конкурентоспособным для случайных наборов точек.

Алгоритм Жарвиса варьируется от 119 до 122 микросекунд, что делает его наиболее медленным среди тестируемых алгоритмов. Несмотря на это, его временная сложность O(nh) делает его подходящим для случаев, где выпуклая оболочка имеет небольшое число вершин (h).**Набор Данных:** равномерно выбрать 100 точек из квадрата [−25;25]×[−25;25]

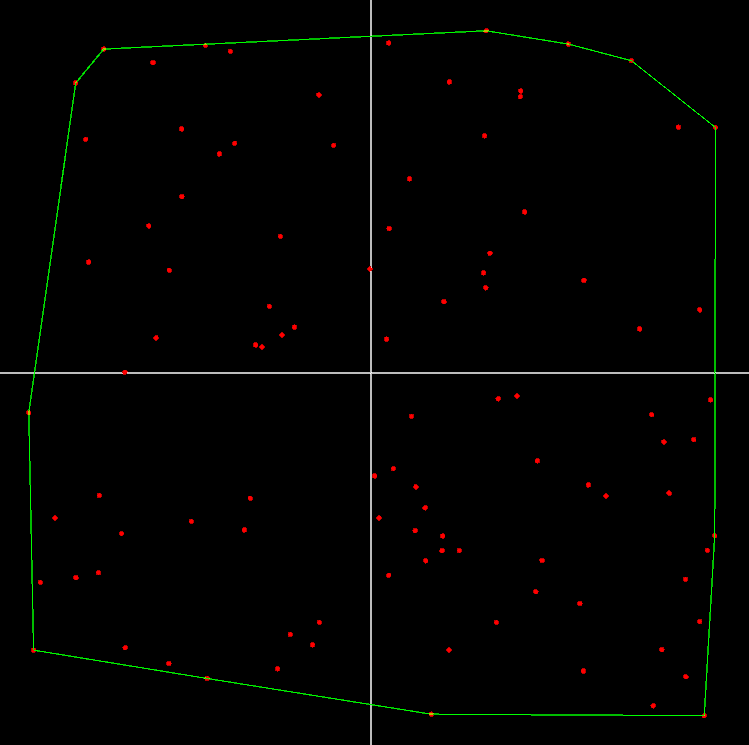


Рисунок 7. Выпуклая оболочка из 100 точек



Рисунок 8. Первый результат для 100 точек



Рисунок 9. Второй результат для 100 точек



Рисунок 10. Третий результат для 100 точек

**Набор Данных:** равномерно выбрать 500 точек из квадрата [−50;50]×[−50;50]

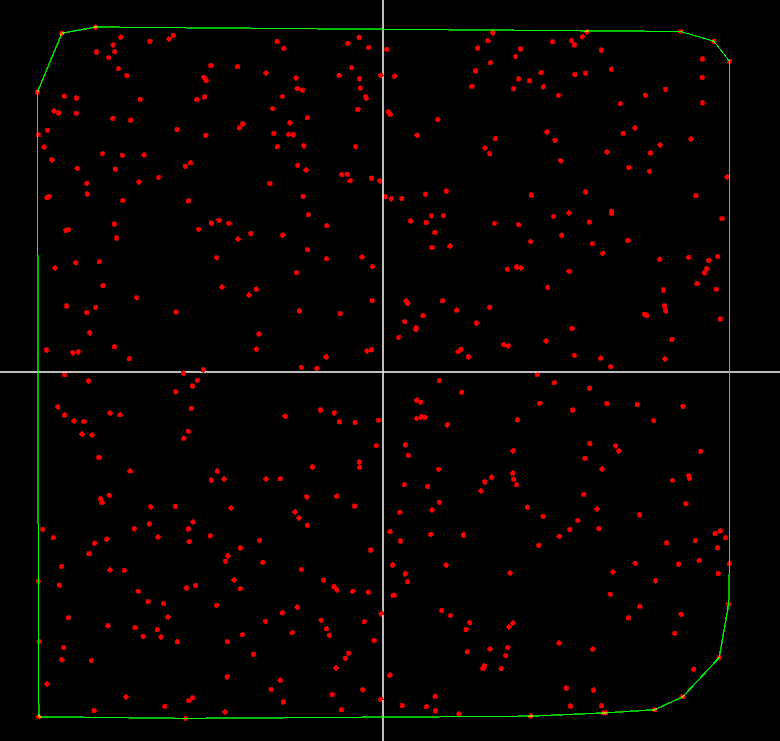


Рисунок 7. Выпуклая оболочка из 500 точек



Рисунок 8. Первый результат для 500 точек



Рисунок 9. Второй результат для 500 точек



Рисунок 10. Третий результат для 500 точек

**Набор Данных:** равномерно выбрать 10000 точек из квадрата [−100;100]×[−100;100]

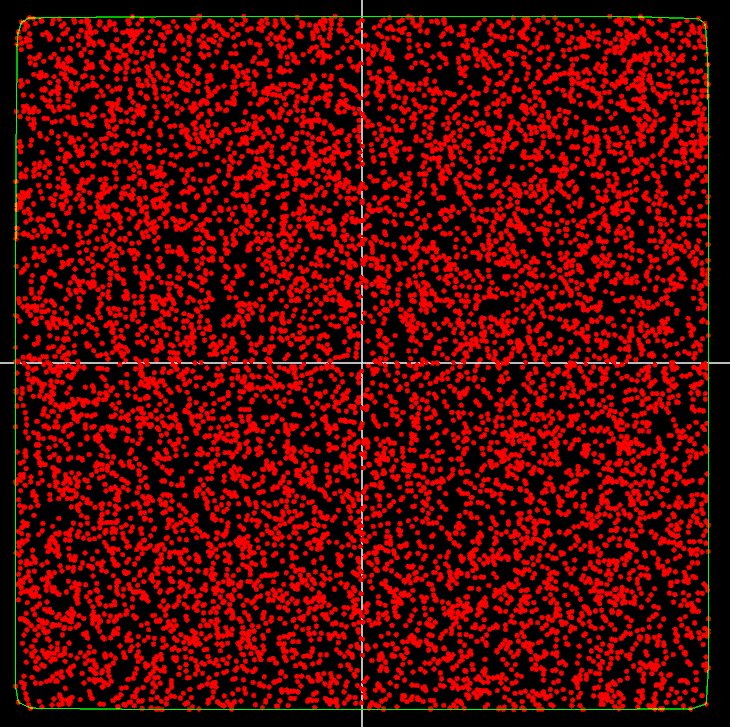


Рисунок 11. Выпуклая оболочка из 10000 точек



Рисунок 8. Первый результат для 10000 точек



Рисунок 9. Второй результат для 10000 точек



Рисунок 10. Третий результат для 10000 точек

# **Заключение**

Алгоритм Грэхэма эффективно решает задачу построения выпуклой оболочки для множества точек на плоскости с временной сложностью , что и Рекурсивный алгоритм. Выбор оптимального алгоритма для построения выпуклой оболочки зависит от характеристик входных данных, таких как размер набора точек и количество вершин выпуклой оболочки. Когда точек для построения выпуклой оболочки немного, Алгоритм Джарвиса рекомендуется использовать. С другой стороны, Алгоритм Грэхэма рекомендуется.

# **Список Литературы**

1. Convex Hull Using Graham Scan – URL: <https://www.geeksforgeeks.org/convex-hull-using-graham-scan/>
2. Выпуклая Оболочка – URL: <https://ru.wikipedia.org/wiki/Выпуклая_оболочка>
3. Построение Минимальных Выпуклых Оболочек – URL: <https://habr.com/ru/articles/144921/>
4. Algorithm and data structures – URL: <https://www.inf.ed.ac.uk/teaching/courses/ads/Lectures/lec16.pdf>

# **Приложения**

#include <iostream>

#include <vector>

#include <set>

#include <random>

#include <chrono>

#include <algorithm>

#include <functional>

#include <fstream>

struct Point

{

float x;

float y;

bool operator<(const Point& other) const

{

return (y < other.y || x < other.x);

}

bool operator==(const Point& other) const

{

return (y == other.y && x == other.x);

}

};

int orientation(Point p, Point q, Point r)

{

// using slope or cross product to value orientation

float val = (q.y - p.y) \* (r.x - p.x) - (r.y - p.y) \* (q.x - p.x);

if (val == 0) return 0; // colinear

return (val > 0) ? 1 : 2; // 1-clockwise, 2-counter-clockwise

}

int findside(Point p, Point q, Point r)

{

float val = -((q.y - p.y) \* (r.x - p.x) - (r.y - p.y) \* (q.x - p.x));

if (val == 0) return 0; // colinear

return (val > 0) ? 1 : -1;

}

float dist(Point p, Point q, Point r)

{

return abs((q.y - p.y) \* (r.x - p.x) - (r.y - p.y) \* (q.x - p.x));

}

// алгоритм Грэхема

std::vector<point> GrahamAlgo(std::vector<point> points) {

size\_t n = points.size();

if (n < 3) return points; // Выпуклая оболочка невозможна, если точки меньше, чем 3

// 1. Найти точку с наименьшей координатой Y (и X в случае tie)

int ymin = 0;

for (int i = 1; i < n; i++) {

if (points[i].y < points[ymin].y ||

(points[i].y == points[ymin].y && points[i].x < points[ymin].x)) {

ymin = i;

}

}

// 2. Поместить эту точку в начало

std::swap(points[0], points[ymin]);

point p0 = points[0];

// 3. Сортировка оставшихся точек по углу с p0

std::sort(points.begin() + 1, points.end(), [&](const point& p1, const point& p2) -> bool {

int o = orientation(p0, p1, p2);

if (o == 0)

return distsq(p0, p1) < distsq(p0, p2);

return o == 2; // против часовой стрелки

});

// 4. Удаление точек с одинаковым углом, оставляя самую дальнюю

std::vector<point> unique\_points;

unique\_points.push\_back(points[0]);

for (int i = 1; i < n; i++) {

// Удаляем все точки, которые коллинеарны с p0 и p[i]

while (i < n - 1 && orientation(p0, points[i], points[i + 1]) == 0)

i++;

unique\_points.push\_back(points[i]);

}

// Если после удаления коллинеарных точек осталось меньше 3, оболочка невозможна

if (unique\_points.size() < 3)

return unique\_points;

// 5. Инициализация стека (вектора) для хранения точек оболочки

std::vector<point> hull;

hull.push\_back(unique\_points[0]);

hull.push\_back(unique\_points[1]);

hull.push\_back(unique\_points[2]);

// 6. Обработка оставшихся точек

for (size\_t i = 3; i < unique\_points.size(); i++) {

// Пока последние три точки не образуют поворот против часовой стрелки

while (hull.size() >= 2 &&

orientation(nextToTop(hull), hull.back(), unique\_points[i]) != 2) {

hull.pop\_back();

}

hull.push\_back(unique\_points[i]);

}

return hull;

}

void quickhull(std::set<Point>& hull, std::vector<Point> points, Point p1, Point p2, int side)

{

int id = -1;

int max\_dist = 0;

for (int i = 0; i < points.size(); i++)

{

float temp = dist(p1, p2, points[i]);

if (findside(p1, p2, points[i]) == side && temp > max\_dist)

{

id = i;

max\_dist = temp;

}

}

if (id == -1)

{

hull.insert(p1);

hull.insert(p2);

return;

}

quickhull(hull, points, points[id], p1, -findside(points[id], p1, p2));

quickhull(hull, points, points[id], p2, -findside(points[id], p2, p1));

}

void recursive\_convexhull(std::set<Point>& hull, std::vector<Point> points)

{

int min\_x = 0;

int max\_x = 0;

for (int i = 1; i < points.size(); i++)

{

if (points[i].x < points[min\_x].x)

{

min\_x = i;

}

if (points[i].x > points[max\_x].x)

{

max\_x = i;

}

}

quickhull(hull, points, points[min\_x], points[max\_x], 1);

quickhull(hull, points, points[min\_x], points[max\_x], -1);

}

std::vector<Point> convex\_hull(std::vector<Point> points)

{

int left\_most\_index = 0;

int n = points.size();

for (int i = 0; i < points.size(); i++)

{

if (points[i].x < points[left\_most\_index].x)

{

left\_most\_index = i;

}

}

int first\_index = left\_most\_index;

std::vector<Point> hull;

Point p = points[left\_most\_index];

hull.push\_back(p);

while (true)

{

p = points[left\_most\_index];

int next\_index = (left\_most\_index + 1) % n;

for (int i = 0; i < points.size(); i++)

{

if (orientation(p, points[next\_index], points[i]) == 2)

{

next\_index = i;

}

}

if (next\_index == first\_index)

{

break;

}

hull.push\_back(points[next\_index]);

left\_most\_index = next\_index;

}

return hull;

}

void points\_generation(std::vector<Point>& points, int num\_point = 1000, float start = -100.0f, float end =

100.0f)

{

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<float> dist(start, end);

for (int i = 0; i < num\_point; i++)

{

float x = dist(gen);

float y = dist(gen);

points.push\_back(Point{ x, y });

}

}

bool load\_points\_from\_file(const std::string& file\_name, std::vector<Point>& points)

{

std::ifstream in\_file(file\_name);

if (!in\_file.is\_open())

{

return false;

}

Point point;

while (in\_file >> point.x >> point.y)

{

points.push\_back(point);

}

in\_file.close();

return true;

}

void save\_points\_to\_file(const std::string& file\_name, const std::vector<Point>& points)

{

std::ofstream out\_file(file\_name);

if (!out\_file.is\_open())

{

std::cerr << "Error: Unable to open file for writing.\n";

return;

}

for (const auto& point : points)

{

out\_file << point.x << " " << point.y << "\n";

}

out\_file.close();

}

int main(int argc, char\*\* argv)

{

//global variable:

std::vector<Point> points;

std::string file\_name = "point.txt";

if (load\_points\_from\_file(file\_name, points))

{

std::cout << "Points loaded from " << file\_name << ".\n";

}

else

{

std::cout << file\_name << " not found. Generating points.\n";

points\_generation(points);

save\_points\_to\_file(file\_name, points);

std::cout << "Points saved to " << file\_name << ".\n";

}

std::vector<Point> hull;

std::set<Point> recursive\_hull;

std::vector<Point> graham\_hull;

points\_generation(points);

//points = { {1, 1}, {2, 5}, {1, 9}, {4, 3}, {6, 4}, {5, 7}, {8, 2}, {8, 9} };

auto start = std::chrono::high\_resolution\_clock::now();

hull = convex\_hull(points);

auto stop = std::chrono::high\_resolution\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(stop - start);

std::cout << "Duration for Jarvis: " << duration.count() << "\n";

auto start2 = std::chrono::high\_resolution\_clock::now();

recursive\_convexhull(recursive\_hull, points);

auto stop2 = std::chrono::high\_resolution\_clock::now();

auto duration2 = std::chrono::duration\_cast<std::chrono::microseconds>(stop2 - start2);

std::cout << "Duration for QuickHull: " << duration2.count() << "\n";

auto start3 = std::chrono::high\_resolution\_clock::now();

graham\_hull = GrahamAlgo(points);

auto stop3 = std::chrono::high\_resolution\_clock::now();

auto duration3 = std::chrono::duration\_cast<std::chrono::microseconds>(stop3 - start3);

std::cout << "Duration for Graham: " << duration3.count() << "\n";

std::cout << "Jarvis results: \n";

for (auto p : hull)

{

std::cout << p.x << " " << p.y << "\n";

}

std::cout << "--------------\n";

std::cout << "Recursive results: \n";

for (auto p1 : recursive\_hull)

{

std::cout << p1.x << " " << p1.y << "\n";

}

std::cout << "--------------\n";

std::cout << "Graham results: \n";

for (auto p2 : graham\_hull)

{

std::cout << p2.x << " " << p2.y << "\n";

}

return 0;

}