Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

«Алгоритм Бентли-Оттманна»

Дисциплина: «Объектно-ориентированное программирование»

Студент гр. 3331506/90401 Мымрин А.В.
Преподаватель Ананьевский М. С.

Санкт-Петербург
2022

ВВЕДЕНИЕ

Вычислительная геометрия – раздел информатики, изучающий алгоритмы, используемые для решения геометрических задач. В данной работе рассмотрен алгоритм Бентли-Оттманна, который, принимая в качестве входных данных множество плоских отрезков, определяет все точки пересечения этих отрезков за время $O((n + k)\log(n))$, где k – количество точек пересечения. Важно отметить, что алгоритм не предполагает наличия вертикальных отрезков в наборе входных данных, также не должно быть пересечения более двух отрезков в одной точке. Данный алгоритм может применяться в робототехнике, а также при разработке игр.

ОПИСАНИЕ АЛГОРИТМА

Принцип действия алгоритма заключается в следующем: вдоль набора отрезков слева направо движется вертикальная прямая, которая называется выметающей. Пересекая отрезки при определенной координате *x*, выметающая прямая помогает упорядочить те отрезки, которые она пересекает, по координате *y*, т.е. по высоте.

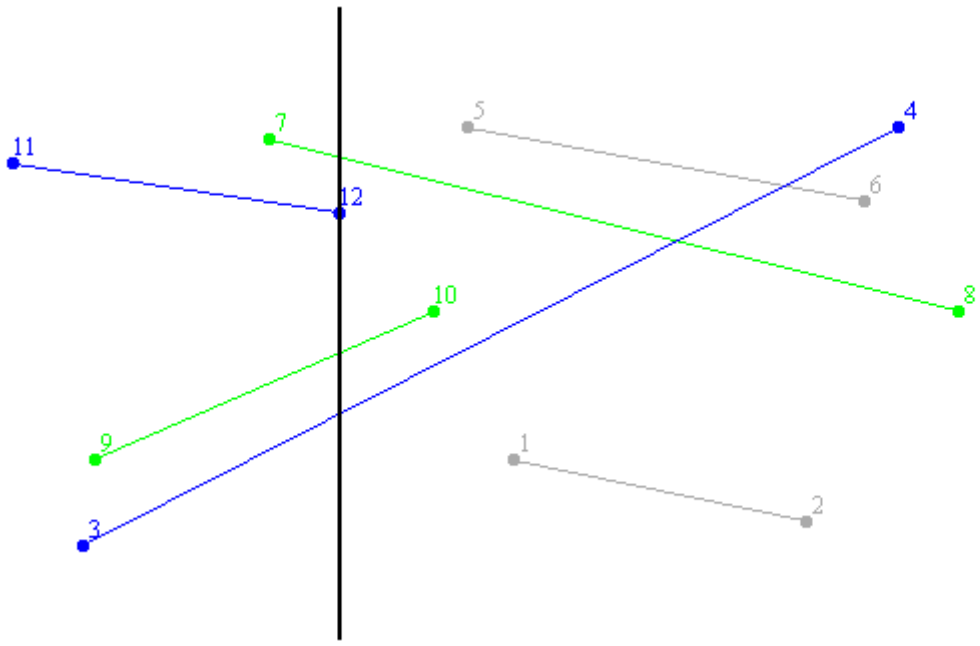

Рисунок 1 – Выметающая прямая

Когда выметающая прямая пересекает начальную точку какого-либо отрезка, он заносится в «очередь» – динамическую структуру данных, в которой по высоте отсортированы все отрезки, которые выметающая прямая пересекает в данный момент. Соответственно, когда прямая достигает конечной точки отрезка, он из очереди удаляется. В случае, когда выметающая прямая находится в точке пересечения двух отрезков, в очереди они меняются местами друг с другом, так как меняется их взаимное расположение.

Так как состояние отрезков относительно выметающей прямой изменяется только в описанных выше случаях, имеет смысл рассматривать состояние очереди только в этих точках. Перед началом работы алгоритма занесем все начальные и конечные точки в другую динамическую структуру данных –

расписание точек-событий. В ней все концы отрезков, а также занесенные туда в будущем точки пересечений отсортированы по координате *x*. Сам алгоритм описывает представленный ниже псевдокод. В нем *Q* – расписание точек-событий, *R* – очередь.

FOREACH point *p* in *Q* (в порядке возрастания x)
DO IF *p* – левая конечная точка отрезка *s*
THEN вставить в *R;*
проверить, пересекает ли s отрезки непосредственно выше и ниже него, в случае, если он пересекает отрезок *t*, вставить точку пересечения s и *t* в *Q* (в порядке x)
ELSE IF *p* – правая конечная точка отрезка s
THEN IF точка пересечения пары отрезков непосредственно выше и ниже *s* не находится в *Q* **THEN** проверить их на пересечение, и если они пересекаются, то добавить их точку пересечения в *Q* (in x-order); удалить отрезок из *R*
ELSE // *p* – точка пересечения отрезков s и *t*, сообщить о паре пересечений; поменять местами позиции s и *t* в *R*
(обратите внимание, что они были и остаются смежными);
проверить верхний отрезок (скажем, s) на пересечение с отрезком над ним, нижний отрезок *(t)* с отрезком под ним, любые точки пересечений добавить в *Q*

Для определения, пересекаются ли два отрезка, используют векторное произведение. Два отрезка пересекаются, если каждый из них пересекает прямую, содержащую другой отрезок.

ИССЛЕДОВАНИЕ АЛГОРИТМА

Для исследования алгоритма с помощью библиотеки `<random>` генерировался случайный набор отрезков. Время работы алгоритма измерялось с помощью библиотеки `<ctime>`. Динамические структуры данных реализованы с помощью библиотеки `<map>` с использованием красно-черных деревьев.

В таблице 1 представлены экспериментальные данные.

Таблица 1 – экспериментальные данные

n, отрезков	t, мс	k, точек пересечения
100	4	26
1000	84	289
5000	1054	1415
7500	1996	2119
10000	3622	2778
15000	5926	4221
20000	8627	5621
25000	11949	7064
30000	16312	8505
35000	20966	9922

На рисунке 2 изображен график зависимости времени работы алгоритма от количества отрезков.

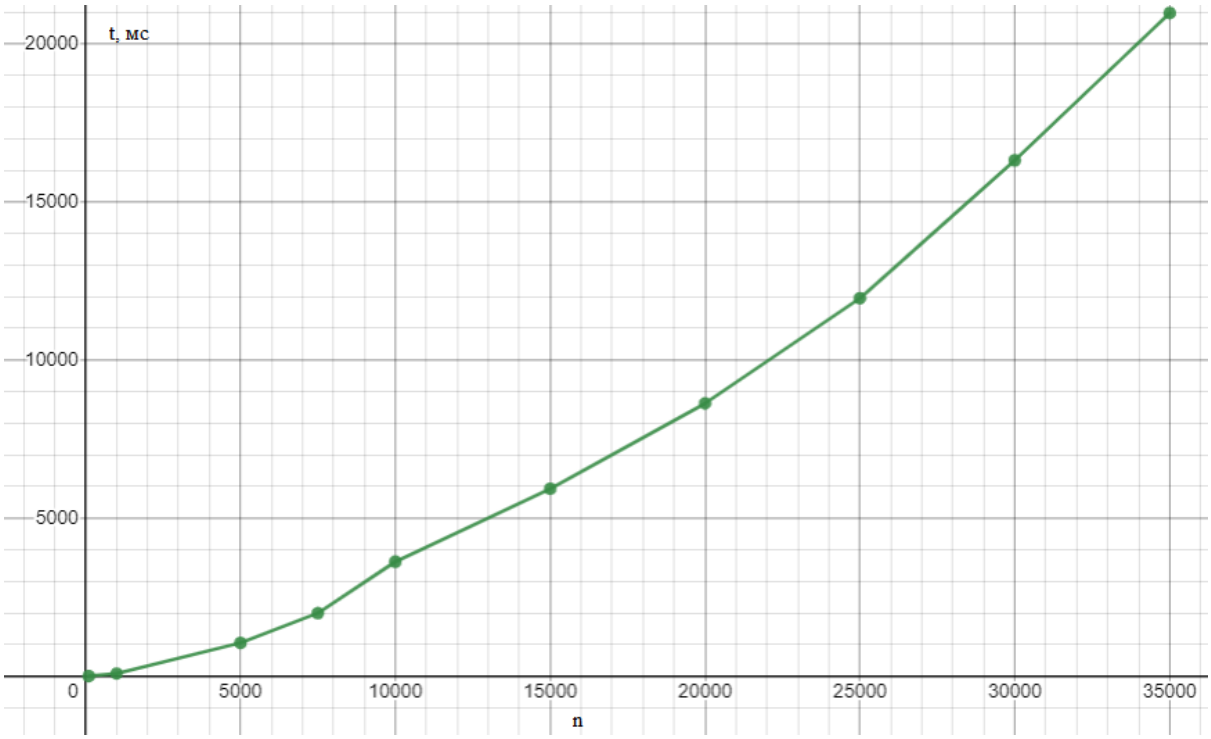

Рисунок 2 – Зависимость времени работы алгоритма от количества отрезков

Как видно из графика, алгоритм имеет сложность $O(n\log(n))$.

В случае, когда отрезки многократно друг с другом пересекаются, время работы алгоритма значительно снижается. В таблице 2 приведены экспериментальные данные.

Таблица 2 – экспериментальные данные при многократных пересечениях

n, отрезков	t, мс	k, точек пересечения
10	1	10
25	19	103
50	233	390
75	641	793
100	1565	1301
150	5597	2926
200	10370	4806
250	20105	7130
300	40536	10624
350	85451	14606
400	144170	18455

На рисунке 3 изображен график эксперимента.

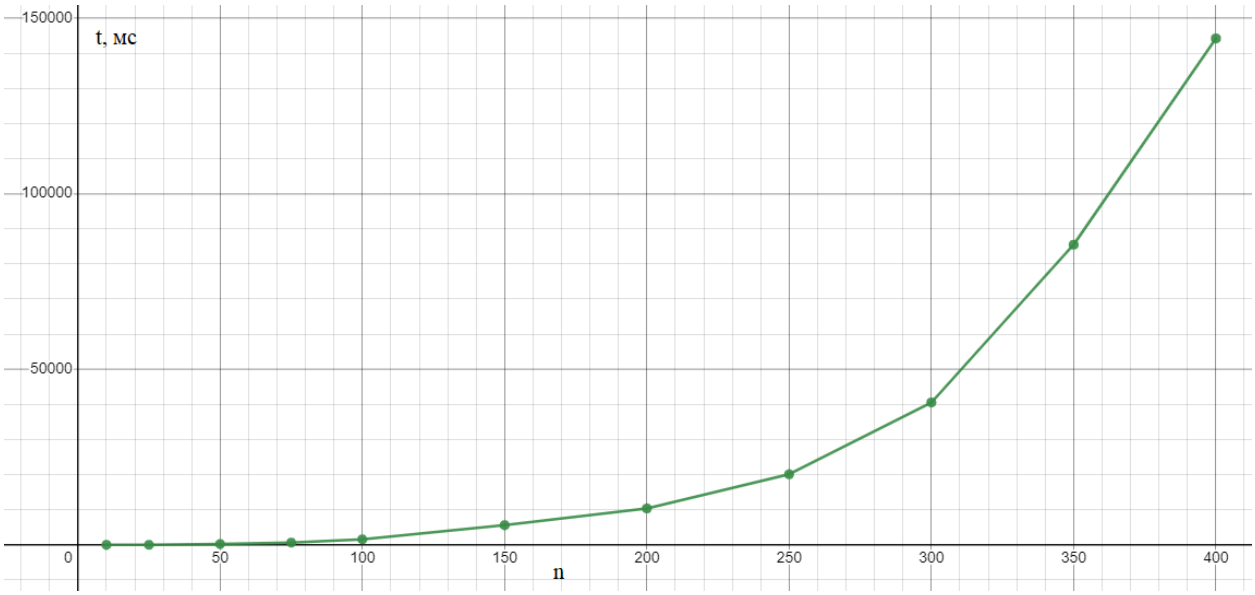

Рисунок 3 – Эксперимент при многократных пересечениях

ЗАКЛЮЧЕНИЕ

Был рассмотрен алгоритм по поиску всех точек пересечений множества отрезков. Результаты экспериментов показали, что алгоритм становится малоэффективен, если отрезки многократно пересекаются. Скорость алгоритма также зависит от реализации динамических структур данных.

СПИСОК ЛИТЕРАТУРЫ

1. Jon L. Bentley, Thomas A. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. // IEEE transactions on computers. –1979. –vol. C-28. pp. 643-647.

2. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы. Построение и анализ — 2-е изд. — “Вильямс”, 2005. — 1296 с.

ПРИЛОЖЕНИЕ 1

Код алгоритма

```
#include <iostream>
#include <cmath>
#include <algorithm>
#include <random>
#include <ctime>
#include <map>
#include "bentley_ottmann.h"

using namespace std;

double eps=0.000000001; //eps=10^-9

Point::Point(double x, double y) {
    this->x=x;
    this->y=y;
}

Point::Point() {
    x=0;
    y=0;
}

Segment::Segment(Point first, Point last) : first(first),last(last) {
    if (this->first.x < this->last.x) {
        this->first=first;
        this->last=last;
    } else {
        this->first=last;
        this->last=first;
    }
    this->cur_x=cur_x;
}

Segment::Segment() {
    first={0,0};
    last={0,0};
    cur x=0;
}

Segment & Segment::operator=(const Segment &s) {
    if (&s== this){
        return *this;
    }
    first.x=s.first.x;
    first.y=s.first.y;
    last.x=s.last.x;
    last.y=s.last.y;
    return *this;
}

//положение вектора ab относительно ac (по часовой(если >0) или против часовой стрелки(если <0))
double Event::direction(Point a, Point b, Point c) {
    return (b.x-a.x)*(c.y-a.y)-(c.x-a.x)*(b.y-a.y);
}

bool operator<(const Segment &s1,const Segment &s2) {

    double dir012=Event::direction(s1.first,s1.last,s2.first);
    double dir302=Event::direction(s2.last,s1.first,s2.first);
    double dir013=Event::direction(s1.first,s1.last,s2.last);
    if(Event::intersect(s1,s2)) {
        double ints_x=Event::intersection_point(s1,s2).x;
        double curx=max(s1.cur x,s2.cur x);
        if (curx<(ints x-eps)) //если x выметающей меньше x точки пересечения
            return ((dir012>0)&&(dir302>0)&&(dir013)>0) or ((dir012>0)&&(dir302<0)&&(dir013)>0) or
((dir012>0)&&(dir302<0)&&(dir013)<0) or ((dir012<0)&&(dir302<0)&&(dir013)>0);
        else
            return !(((dir012>0)&&(dir302>0)&&(dir013)>0) or ((dir012>0)&&(dir302<0)&&(dir013)>0) or
((dir012>0)&&(dir302<0)&&(dir013)<0) or ((dir012<0)&&(dir302<0)&&(dir013)>0));
    }
    return ((dir012>0)&&(dir302>0)&&(dir013)>0) or ((dir012>0)&&(dir302<0)&&(dir013)>0) or
((dir012>0)&&(dir302<0)&&(dir013)<0) or ((dir012<0)&&(dir302<0)&&(dir013)>0);
}
```

```
bool operator==(const Segment &s1, const Segment &s2) {
    return (fabs(s1.first.x-
s2.first.x)<eps)&&(fabs(s1.first.y==s2.first.y)<eps)&&(fabs(s1.last.x==s2.last.x)<eps)&&(fabs(s1.las
t.y==s2.last.y)<eps);
}

default random engine generator;
uniform_real_distribution<double> distribution(0.1,90);
double get_random() {
    return distribution(generator);
}

bool Event::on_segment(Point a, Point b, Point c) { //определяет, лежит ли точка c на отрезке ab
    if (fabs(direction(a,b,c))<eps and ((c.x>(a.x+eps))&&(c.x<(b.x-eps)) or
(c.x>(b.x+eps))&&(c.x<(a.x-eps)))and ((c.y>(a.y+eps))&&(c.y<(b.y-eps)) or
(c.y>(b.y+eps))&&(c.y<(a.y-eps))))
        return true;
    return false;
}

bool Event::intersect(Segment a, Segment b) { //определяет, есть ли пересечение
    double dir1= direction(b.first,b.last,a.first);
    double dir2= direction(b.first,b.last,a.last);
    double dir3= direction(a.first,a.last,b.first);
    double dir4= direction(a.first,a.last,b.last);
    if (((dir1>0)&&(dir2<0)or(dir1<0)&&(dir2>0))and((dir3>0)&&(dir4<0)or(dir3<0)&&(dir4>0)))
        return true;
    if (on segment(b.first,b.last,a.first))
        return true;
    if (on_segment(b.first,b.last,a.last))
        return true;
    if (on_segment(a.first,a.last,b.first))
        return true;
    if (on segment(a.first,a.last,b.last))
        return true;
    return false;
}

Point Event::intersection_point(Segment a, Segment b){ // Точка пересечения из уравнения прямой по
двум ее точкам
    double x1=a.first.x;
    double x2=a.last.x;
    double x3=b.first.x;
    double x4=b.last.x;

    double y1=a.first.y;
    double y2=a.last.y;
    double y3=b.first.y;
    double y4=b.last.y;

    double ax=(y2-y1)/(x2-x1);
    double bx=(y4-y3)/(x4-x3);
    double ay=(x2-x1)/(y2-y1);
    double by=(x4-x3)/(y4-y3);

    double intersec_x=(x1*ax-x3*bx+y3-y1)/(ax-bx);
    double intersec y=(y1*ay-y3*by+x3-x1)/(ay-by);
    return {intersec_x,intersec_y};

}

multimap <double,Segment> make schedule(Segment data[],multimap <double,Segment> schedule, int size)
{
    for (int i=0;i<size;i++) {
        schedule.insert(make_pair(data[i].first.x,data[i]));
        schedule.insert(make pair(data[i].last.x,data[i]));
    }
    return schedule;
}
//существует ли в структуре отрезок выше
bool below q( multimap <Segment, Segment> :: iterator it, multimap <Segment,Segment> &queue) {
    return it!=queue.begin();
}
//существует ли в структуре отрезок ниже
bool above_q(multimap <Segment, Segment> :: iterator it, multimap <Segment,Segment> &queue) {
    it++;
    return it!=queue.end();
}
//определяет, есть ли такая же точка пересечения в контейнере
```

```
bool equal intpoint(multimap <double,pair<Segment,Segment>> intersections,double value) {
    multimap <double,pair<Segment,Segment>> :: iterator it;
    it=intersections.find(value);
    if (it!=intersections.begin()) {
        it--;
        if(fabs(value-it->first)<eps)
            return true;
        it++;
    }
    if ((it++)!=intersections.end()) {
        if (fabs(value-it->first)<eps)
            return true;
    }
    return false;
}

int intp cnt=0; //счетчик точек пересечения

void event_handler(multimap <double,Segment> schedule,multimap <Segment,Segment> queue) {
    multimap <double,pair<Segment,Segment>> intersections;
    auto qit=queue.begin(); //итератор для очереди

    for(auto &it : schedule) {
        Event event{};
        Point intpoint{};
        if(intersections.find(it.first)!=intersections.end()) { //если x найдено в контейнере точек
пересечения

            Segment above=intersections.find(it.first)->second.first;
            Segment below=intersections.find(it.first)->second.second;
            qit=queue.find(above);//итератор на верхний отрезок
            queue.erase(qit);
            above.cur_x=it.first;
            queue.insert(make pair(above,above)); //замена верхнего и нижнего отрезка на новые с
новым значением cur x

            qit=queue.find(below);
            queue.erase(qit);
            below.cur x=it.first;
            queue.insert(make_pair(below,below)); //новые отрезки будут иметь уже другой порядок
относительно друг друга

            qit=queue.find(below);

            //далее проверим, есть ли точки пересечения с соседями у этих отрезков
            //т.е. верхнего отрезка с соседом сверху и нижнего с соседом снизу

            if (above_q(qit,queue)) { //если итератор - не начало контейнера - есть что-то выше
                qit++;
                Segment above plus = qit->second; //если есть отрезок выше, инициализируем его
                qit--;
                if(event.intersect(qit->second,above_plus)) { //если отрезки пересекаются
                    intpoint=event.intersection_point(qit->second,above_plus);//ищем точку
пересечения
                    intersections.insert(make pair(intpoint.x, make pair(above plus,qit->second)));
                    if (equal intpoint(intersections,intpoint.x)) {
                        intersections.erase(intersections.find(intpoint.x));
                    } else {
                        intp_cnt++;
                        schedule.insert(make pair(intpoint.x,above plus));// внесем в расписание
отрезок
                    }
                }
            }
            qit=queue.find(above);
            if(below_q(qit,queue)) { //существует ли отрезок ниже
                qit--;
                Segment below plus=qit->second;
                below.cur_x=it.first;
                qit++;
                if(event.intersect(qit->second,below_plus)) { //если отрезки пересекаются
                    intpoint=event.intersection point(qit->second,below plus);//ищем точку
пересечения
                    intersections.insert(make_pair(intpoint.x, make_pair(qit->second,below_plus)));
                    if (equal_intpoint(intersections,intpoint.x)) {
                        intersections.erase(intersections.find(intpoint.x));
                    } else {
                        intp_cnt++;
                        schedule.insert(make pair(intpoint.x,below plus));// внесем в расписание
отрезок
```

```
                    }
                    qit--;
                }
            }
        } else
        if (it.first==it.second.first.x) { // если точка НАЧАЛЬНАЯ, вносим в очередь
            it.second.cur_x=it.first; //x выметающей прямой запишем в cur_x отрезка
            queue.insert(pair<Segment,Segment>(it.second,it.second)); //вносим в очередь
            qit=queue.find(it.second); //итератор на отрезок, которому принадлежит точка
            if(above_q(qit,queue)){ //если итератор - не начало контейнера - есть что-то выше
                qit++; //итератор на отрезок выше
                Segment above = qit->second; //если есть отрезок выше, инициализируем его
                qit--;
                if(event.intersect(qit->second,above)) { //если отрезки пересекаются
                    intpoint=event.intersection_point(qit->second,above);//ищем точку пересечения
                    intersections.insert(make_pair(intpoint.x, make_pair(above,qit->second)));
                    if (equal intpoint(intersections,intpoint.x)) { //если такая тока пересечения
уже есть
                        intersections.erase(intersections.find(intpoint.x)); //удаляем из контейнера
точек пересечения
                    } else {
                        intp cnt++;
                        schedule.insert(make_pair(intpoint.x,above));// внесем в расписание отрезок
                    }
                }
            }
            if(below q(qit,queue)) { //существует ли отрезок ниже
                qit--;
                Segment below=qit->second;
                below.cur x=it.first;
                qit++;
                if(event.intersect(qit->second,below)) { //если отрезки пересекаются
                    intpoint=event.intersection_point(qit->second,below);//ищем точку пересечения
                    intersections.insert(make pair(intpoint.x, make pair(qit->second,below)));
                    if (equal intpoint(intersections,intpoint.x)) {
                        intersections.erase(intersections.find(intpoint.x));
                    } else {
                        intp_cnt++;
                        schedule.insert(make pair(intpoint.x,below));// внесем в расписание отрезок
                    }
                    qit--;
                }
            }
        } else
        if (it.first==it.second.last.x) {//если точка - КОНЕЦ
            it.second.cur x=it.first; //записываем в отрезок x выметающей прямой
            qit=queue.find(it.second); //итератор на данный отрезок в очереди

            //если есть отрезки выше и ниже, проверяем их на пересечение

            if(above q(qit,queue) && below q(qit,queue)) {
                qit++;
                Segment above =qit->second; //определим верхний отрезок
                above.cur_x=it.first;
                advance(qit,-2);
                Segment below=qit->second; //определим нижний отрезок
                below.cur_x=it.first;
                qit++;
                if(event.intersect(above,below)) {
                    intpoint=event.intersection point(above,below);//ищем точку пересечения
                    intersections.insert(make_pair(intpoint.x, make_pair(above,below)));
                    if (equal intpoint(intersections,intpoint.x)) {
                        intersections.erase(intersections.find(intpoint.x));
                    } else {
                        intp cnt++;
                        schedule.insert(make_pair(intpoint.x,above));// внесем в расписание отрезок
                    }
                }
            }
            queue.erase(qit); //обработав конец, отрезок можно удалить из очереди
        }
    }
}

void seg_inp(Segment s[],int numb) { //вводим рандомные отрезки
    for (int i=0;i<numb;i++) {
        Point p;
        Point q;
        p=Point{get random(),get random()};
        q=Point{get_random(),get_random()};
```

```
        s[i]=Segment{p,q};
    }
}

int main() {

    static Segment s[30];

    multimap <double,Segment> schedule;
    multimap <Segment,Segment> queue;

    seg inp(s,30);
    schedule=make schedule(s,schedule,30);
    event_handler(schedule,queue);
    cout << "count: "<<intp_cnt<<endl;
    cout << "runtime = " << clock() <<"ms"<< endl;
    return 0;
}
```