ГЕОМЕТРИЯ

Школа::Кода

Олимпиадное программирование

2020-2021 Таганрог

Точки и вектора

- Любую геометрическую задачу можно решить оперируя только точками и векторами.
- Отрезок можно представить как 2 точки. Ломаную – как последовательный набор точек. Любой многоугольник – это замкнутая ломаная.
- Прямую или луч можно представить как точку и направляющий вектор.
- Окружность или сферу – как точку, задающую центр, и радиус-вектор.
- Для решения задач определяются структуры, описывающие точку и вектора, после чего реализуется алгоритм, который уже не содержит в себе координатного метода.

Реализация вектора 1

- Создадим структуру Vector. Сделаем её шаблонной так, как некоторые задачи решаются в целых числах, а некоторые в вещественных. Шаблонность добавляет универсальности но обычно не нужна в условиях соревнования.

```
template<class Type>
struct Vector
{
    Type x;
    Type y;
    Vector(Type x, Type y)
        :x(x), y(y)
    {}
    Vector() {}
```

Реализация вектора 2

- Одной из характеристик вектора является его длина, которая равна sqrt(x^2 + y^2). Заметим, что даже если сам вектор имеет целочисленные координаты, его длина, скорее всего, имеет вещественное значение. Однако в некоторых задачах достаточно узнавать квадрат длины.

```
double len() const
{
    return sqrt(double(x * x + y * y));
}

Type len_sqr() const
{
    return (x * x + y * y);
}
```

- Слово const после имени метода позволяет его вызывать у констант типа данной структуры.

Реализация вектора 3

- Сумма двух векторов – это вектор начало которого совпадает с началом первого, а конец – с концом второго, отложенного от конца первого.
- Разность двух векторов – это вектор, который начинается в конце второго и заканчивается в конце первого, если первый и второй отложить от одной точки.

```
friend Vector operator+(const Vector& a, const Vector& b)
{
    return Vector(a.x + b.x, a.y + b.y);
}

friend Vector operator- (const Vector& a, const Vector& b)
{
    return Vector(a.x - b.x, a.y - b.y);
}

friend Vector operator- (const Vector& a)
{
    return Vector(-a.x, -a.y);
}
```

- Унарный минус меняет направление вектора на обратное.

Реализация вектора 4

- Скалярное произведение векторов a и b – это число, равное произведению длин этих векторов на косинус угла между ними. Зная координаты векторов, скалярное произведение можно вычислить как a.x * b.x + a.y * b.y в двухмерном пространстве или a.x * b.x + a.y * b.y + a.z * b.z в трёхмерном пространстве.
- Векторное произведение векторов a и b в трёхмерном пространстве – это вектор с направленный перпендикулярно плоскости, задаваемой данными векторами, длина которого равна произведению длин этих векторов на синус угла между ними. В двухмерном пространстве векторное произведение – это число, равное a.x * b.y - a.y * b.x. По знаку этого числа можно определить, как по отношению к вектору a повёрнут вектор b (векторное произведение > 0 – против часовой стрелки, < 0 – по часовой стрелке, 0 – коллинеарные).

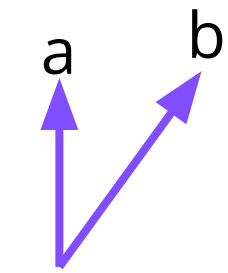
Реализация вектора 4

- Произведения вектора a на число b – это вектор c, коллинеарный вектору a, его длина равна длине a, умноженной на b, а направление совпадает с a, если k > 0, иначе является противоположным. Для того чтобы получить произведение вектора на число нужно каждую координату вектора умножить на это число.

```
friend Type operator* (const Vector& a, const Vector& b)
{
    return a.x * b.x + a.y * b.y;
}

friend Type operator^ (const Vector& a, const Vector& b)
{
    return a.x * b.y - a.y * b.x;
}

friend Vector operator* (const Vector& a, Type b)
{
    return Vector(a.x * b, a.y * b);
}
```



a^b < 0
b^a > 0

Реализация вектора 5

- Для удобства будет не лишним переопределить и операторы потокового ввода-вывода.
- Деление вектора a на число b – это операция, эквивалентная произведению вектора a на число 1/b.

```
friend istream& operator >> (istream& in, Vector& p)
{
    return in >> p.x >> p.y;
}

friend ostream& operator << (ostream& out, const Vector& p)
{
    return out << p.x << ' ' << p.y;
}

friend Vector operator/ (const Vector& a, const Type b)
{
    return Vector(a.x / b, a.y / b);
}
```

Реализация вектора 6

- В качестве вспомогательных шагов часто может понадобиться получить вектор в вещественных числах (если изначально он в целых), нормализовать вектор (получить вектор длиной 1 и с тем же направлением) и изменить длину на заданную.

```
Vector<double> ToFloat() const
{
    return Vector<double>(double(x), double(y));
}

Vector<double> ToNormal() const
{
    return ToFloat() / len();
}

Vector<double> Relen(double l) const
{
    return ToNormal() * l;
}
```

Реализация вектора 7

- Для того что бы повернуть вектор на определённый угол нужно умножить матрицу поворота на этот вектор.

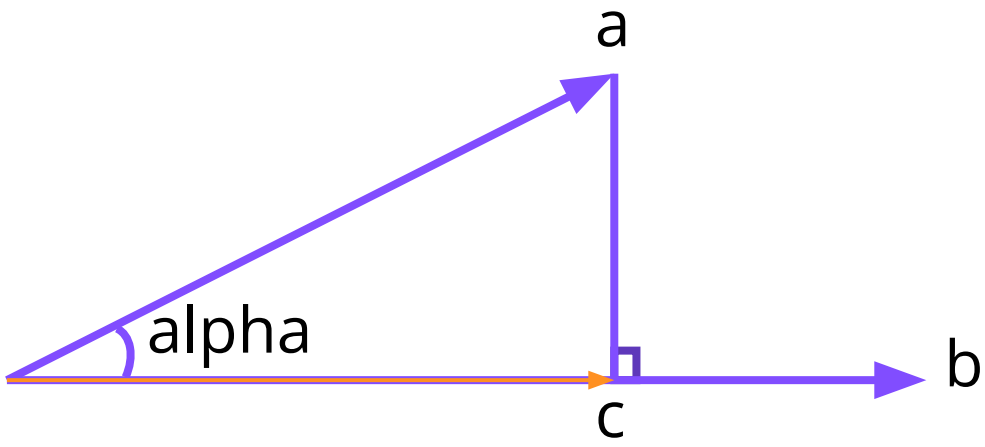
$$\begin{vmatrix} X` \\ Y` \end{vmatrix} = \begin{vmatrix} \mathbf{cos}(a) & \mathbf{sin}(a) \\ \mathbf{-sin}(a) & \mathbf{cos}(a) \end{vmatrix} * \begin{vmatrix} X \\ Y \end{vmatrix} =$$

$$= \begin{vmatrix} X * \mathbf{cos}(a) + Y * \mathbf{sin}(a) \\ -X * \mathbf{sin}(a) + Y * \mathbf{cos}(a) \end{vmatrix}$$

```
Vector Turn(double a)
{
    return Vector<double>
        (sin(a) * y + cos(a) * x,
        cos(a) * y - sin(a) * x);
}
```

- Стандартные функции C++ работают с радианами.

Проекция вектора на вектор



- Вектор c – проекция вектора a на вектор b.
- Длина вектора c равна произведению длины вектора a на косинус угла alpha. Она относиться к длине b как некоторый коэффициент k = |a|*cos(alpha)/|b|.
- Тогда c = b * k.
- Если вспомнить, что a * b = |a| * |b| * cos(alpha), то можно определить k как a * b / $|b|^2$.

Реализация вектора 8

- Метод ProjectionK возвращает коэффициент, на который нужно умножить вектор, у которого вызван метод, чтобы получить проекцию на него вектора, который является аргументом.
- Метод Projection возвращает вектор-проекцию вектора v1 на вектор, у которого вызван метод.

```
double ProjectionK(const Vector& v1) const
{
    return (ToFloat() * v1.ToFloat()) / len_sqr();
}

Vector<double> Projection(const Vector& v1) const
{
    return ToFloat() * ProjectionK(v1);
}
```

Реализация точки 1

- Структура точки довольно похожа на структуру вектора. В некоторых случаях из точки может понадобиться получить вектор. Операторы ввода-вывода переопределяются аналогичным образом.

```
template<class Type>
struct Point
{
    Type x;
    Type y;
    Point(Type x, Type y)
        :x(x), y(y)
    {}
    Point() {}

    Vector<Type> ToVector() const
    {
        return Vector<Type>(x, y);
    }
```

```
    friend istream& operator >> (istream& in, Point& p)
    {
        return in >> p.x >> p.y;
    }

    friend ostream& operator << (ostream& out, const Point& p)
    {
        return out << p.x << " " << p.y;
    }
```

Реализация точки 2

- При прибавлении к точке вектора получается новая точка.
- Соответственно, при вычитании из одной точки другой получается вектор.

```
friend Vector<Type> operator- (const Point& a, const Point& b)
{
    return Vector<Type>(a.x - b.x, a.y - b.y);
}

friend Point operator+ (const Point& a, const Vector<Type>& b)
{
    return Point(a.x + b.x, a.y + b.y);
}
```

- В рамках соревнований, когда времени мало, а готовом шаблоном воспользоваться нельзя, большинство предпочитает реализовывать только структуру вектора в той степени, в какой необходимо, т.к. она по сути полностью покрывает возможности структуры точки.

Нахождение угла между двумя векторами

- Разделив векторное произведение векторов на произведение их длин получим синус угла между ними.
- Разделив скалярное произведение векторов на произведение их длин получим косинус угла между ними.

```
template<typename T>
double get_sin(Vector<T>& a, Vector<T>& b)
{
    return (a ^ b) / a.len() / b.len();
}

template<typename T>
double get_cos(Vector<T>& a, Vector<T>& b)
{
    return (a * b) / a.len() / b.len();
}
```

Немного прекода от себя…

```
typedef long long int64;
typedef unsigned long long uint64;
typedef unsigned int uint;

#define mp make_pair
#define sqr(a) ((a)*(a))
#define forn(i, n) for(int i = 0; i < int(n); ++i)
#define forr(i, n) for(int i = int(n) - 1; i >= 0; --i)
#define forit(i,c) for(auto i=(c).begin(); i!=(c).end(); ++i)
#define ALL(x) (x).begin(), (x).end()

#define MIN(a, b) ((a)<(b)?(a):(b))
#define MAX(a, b) ((a)>(b)?(a):(b))

typedef vector<int> vint;
typedef vector<bool> vbool;
typedef vector<int64> vint64;
typedef vector<uint64> uvint64;
typedef pair<int, int> pii;
typedef pair<int64, int64> pii64;
```

```
typedef Point<int64> Pint;
typedef Point<double> Pdbl;
typedef Vector<int64> Vint;
typedef Vector<double> Vdbl;
typedef pair<Pdbl, Vdbl> Line;
const double Pi = 3.14159265358979323846264338327795;
const double EPS = 1e-9;
```

Полярный угол точки

- Полярный угол точки – это угол между вектором, с началом в точке пересечения координатных осей и концом в этой точке, и осью X.
- Самая точная функция для нахождения полярного угла – это atan2(double Y, double X). Она возвращает значения в диапазоне [-π; π).

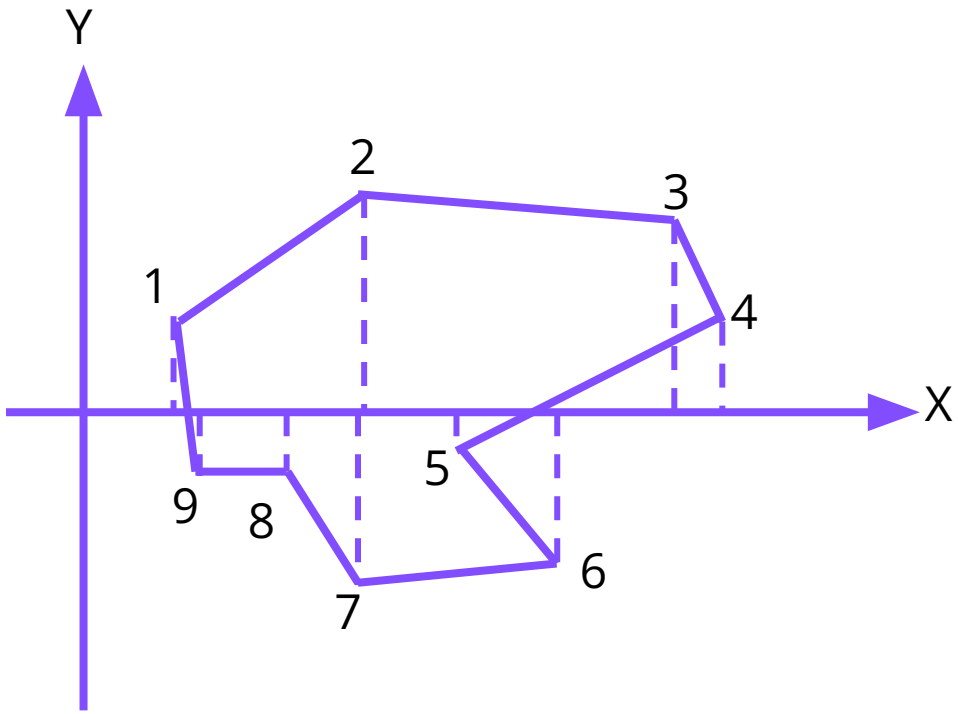
Сравнение вещественных чисел на равенство / неравенство

- Так как вычисления в вещественных числах происходят с некоторой погрешностью, нельзя гарантировать, что результат вычислений будет равен ожидаемому числу со 100% точностью, а значит эти числа нельзя сравнивать через ‘==‘ или ‘!=‘.
- Корректная проверка вещественных чисел на равенство: fabs(a - b) <= EPS (fabs(a - b) > EPS – проверка на неравенство).

Алгоритм нахождения ориентированной площади многоугольника

- Пусть даны вершина N-угольника в порядке обхода.
- Заведём переменную res, изначально равную нулю.
- Переберём все вершины. На каждом шаге будем рассматривать ребро между вершинами i и i+1 прибавлять к результату удвоенную площадь трапеции, образуемой данным ребром, нормалями к оси X и самой осью: res += (p_{i+1}.x - p_i.x) * (p_{i+1}.y + p_i.y).
- В конце возьмём результат по модулю и разделим пополам, т.к. изначально суммировали удвоенную площадь. Это делается для увеличения точности подсчётов. Если координаты точек даны целыми числами, то до последнего шага вычисления можно проводить не прибегая к вещественным числам.

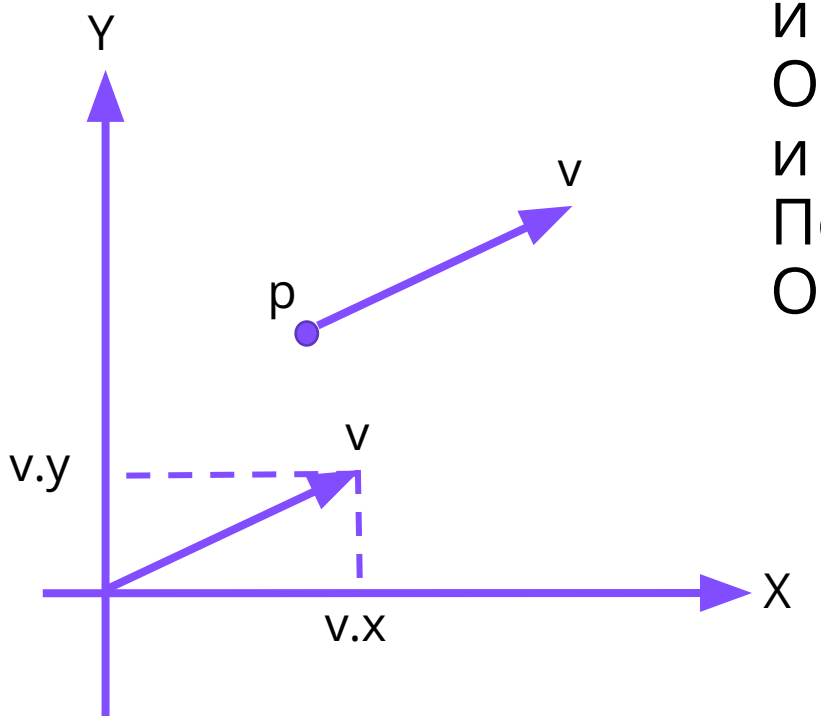
Алгоритм нахождения ориентированной площади многоугольника




```
template<typename T>
double get_area(vector<Point<T>>& p)
{
    T res = 0;
    int n = p.size();
    for (int i = 0; i < n; ++i)
        res += (p[(i + 1) % n].x - p[i].x)
        * (p[(i + 1) % n].y - p[i].y);
    return abs(res) / 2.0;
}
```

Уравнение прямой

- Каноническое уравнение прямой имеет вид a*x + b*y + c = 0.
- Пусть известны коэффициенты уравнения задающего прямую, и нужно найти точку и направляющий вектор для описания этой прямой.

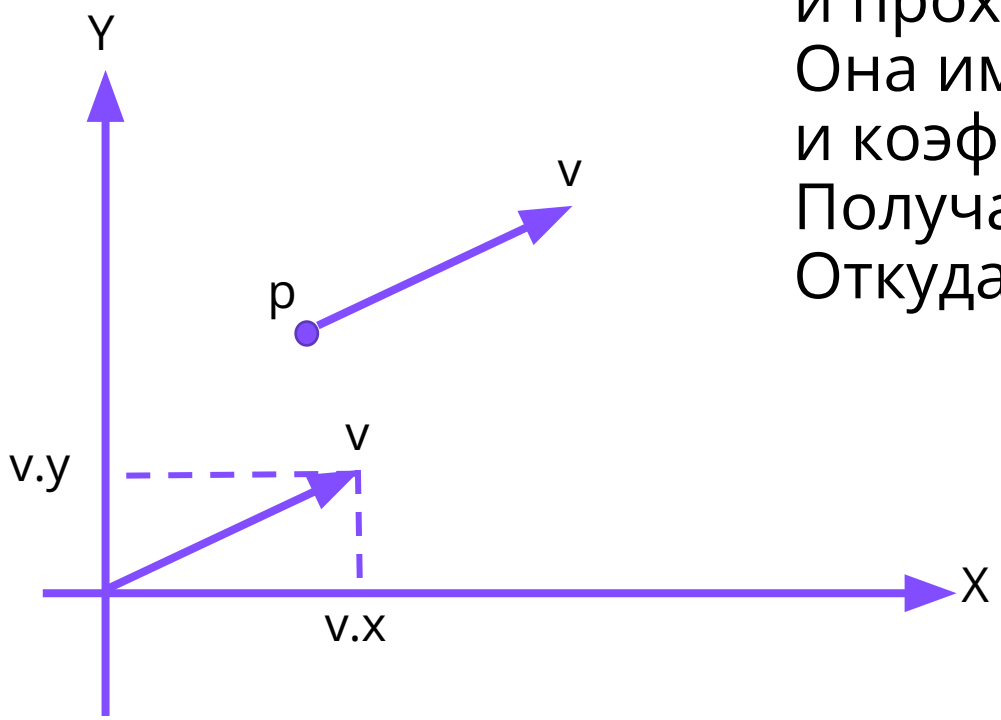


Рассмотрим прямую, параллельную заданной и проходящую через начало координат.
Она имеет такой же направляющий вектор v и коэффициент c = 0.
Получаем, что a*v.x + b*v.y = 0
Откуда v.x = b и v.y = -a

Если прямая не вертикальная (b != 0), то в качестве p.x возьмём 0, тогда уравнение прямой принимает вид b*p.y + c = 0, откуда p.y = -c/b.
Иначе возьмём 0 в качестве p.y, тогда p.x = -c/a

Уравнение прямой 2

- Пусть известна точка p и направляющий вектор v, задающие прямую. Требуется найти коэффициенты канонического уравнения для этой прямой.



Рассмотрим прямую, параллельную заданной и проходящую через начало координат.
Она имеет такой же направляющий вектор v и коэффициент c = 0.
Получаем, что a*v.x + b*v.y = 0
Откуда a = -v.y и b = v.x

Подставим в уравнение
a*x + b*y + c = 0 коэффициенты a, b и координаты p, получим:
-v.y*p.x + v.x*p.y + c = 0, откуда
c = p.x*v.y - p.y*v.x = p.ToVector() ^ v

Реализация функций

```
Line ToLine(double a, double b, double c)
{
    Line l;
    l.second.y = -a;
    l.second.x = b;
    if (b != 0)
    {
        l.first.x = 0;
        l.first.y = -c / b;
    }
    else
    {
        l.first.x = -c / a;
        l.first.y = 0;
    }
    return l;
}
```

```
template<typename T>
void PrintABC(const Point<T>& p, const Vector<T>& v)
{
    double a, b, c;
    a = -v.y;
    b = v.x;
    c = (p.ToVector() ^ v);
    printf("%.12f %.12f %.12f\n", a, b, c);

}
```