**Федеральное государственное автономное образовательное учреждение высшего образования**

**«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

# ОТЧЕТ

*по семестровой работе*

“Построение выпуклой оболочки. Алгоритм Джарвиса”

Группа № 11-002

Кузнецова Александра

Казань – 2021 г.

## Введение

**Алгоритм Джарвиса** (или алгоритм обхода Джарвиса, или алгоритм заворачивания подарка) определяет последовательность элементов множества, образующих выпуклую оболочку для этого множества. Метод можно представить как обтягивание верёвкой множества вбитых в доску гвоздей. Алгоритм работает за время O ( n h ), где n — общее число точек на плоскости, h — число точек в выпуклой оболочке. По-другому "Gift wrapping algorithm" (Заворачивание подарка). Он заключается в том, что мы ищем выпуклую

оболочку последовательно, против часовой стрелки, начиная с определенной точки.
*Выпуклой оболочкой* множества X называется наименьшее выпуклое множество, содержащее множество X.

## Описание алгоритма

Пусть дано множество точек $P = \{ p_1 , p_2 , \dots , p_n \}$ . В качестве начальной берётся самая левая нижняя точка $p_1$ (её можно найти за $O ( n )$ обычным проходом по всем точкам), она точно является вершиной выпуклой оболочки. Следующей точкой ( $p_2$ ) берём такую точку, которая имеет наименьший положительный полярный угол относительно точки $p_1$ как начала координат. После этого для каждой точки $p_i$ $(2<i<=|P|)$ против часовой стрелки ищется такая точка $p_{i+1}$, путём нахождения за $O ( n )$ среди оставшихся точек (+ самая левая нижняя), в которой будет образовываться наибольший угол между прямыми $p_{i-1} p_i$ и $p_i p_{i+1}$ . Она и будет следующей вершиной выпуклой оболочки. Сам угол при этом не ищется, а ищется только его косинус через скалярное произведение между лучами $p_{i-1} p_i$ и $p_i p_{i+1}$ , где $p_i$ — последний найденный минимум, $p_{i-1}$ — предыдущий минимум, а $p_{i+1}$ — претендент на следующий минимум. Новым минимумом будет та точка, в которой косинус будет принимать наименьшее значение (чем меньше косинус, тем больше его угол). Нахождение вершин выпуклой оболочки продолжается до тех пор, пока $p_{i+1} \neq p_1$ . В тот момент, когда следующая точка в выпуклой оболочке совпала с первой, алгоритм останавливается — выпуклая оболочка построена.
Данный алгоритм основан на следующем утверждении. Отрезок L, определяемый двумя точками, является ребром выпуклой оболочки тогда и только тогда, когда все другие элементы множества S лежат на L или с одной стороны от него.

Из этого «вытекает» следующая логика: $N$ точек определяет порядка $n^2$ отрезков. Для каждого из этих отрезков можно определить положение остальных ($N$-2) точек относительно него.

Таким образом, за время порядка $O(N^3)$ можно найти все пары точек, определяющих ребра выпуклой оболочки. Затем эти точки следует расположить в порядке последовательности вершин оболочки.

Джарвис заметил, что данную идею можно улучшить, если учесть следующий факт. Если установлено, что отрезок $q_1q_2$ является ребром оболочки, то должно существовать другое ребро с концом в точке $q_2$, принадлежащее выпуклой

оболочке. Уточнение этого факта приводит к алгоритму со временем работы $O(N^2)$.

## Анализ

Цикл (4) выполнится h раз, при этом цикл (a) каждый раз выполняется за Θ ( n ) . Все остальные операции выполняются за O ( 1 ) . Следовательно, алгоритм работает за Θ ( h n ) или Θ ( n 2 ) в худшем случае, когда в выпуклую оболочку попадут все точки.

## Корректность

Точка $p$ 0 , очевидно, принадлежит оболочке. На каждом последующем шаге алгоритма мы получаем прямую $p_{i-1}$ $p_i$ , по построению которой все точки множества лежат слева от нее. Значит, выпуклая оболочка состоит из $p_i$ -ых и только из них.

## Сложность

Добавление каждой точки в ответ занимает $O(n)$ времени, всего точек будет $k$ , поэтому итоговая сложность $O(nk)$. В худшем случае, когда оболочка состоит из всех точек сложность $O(n_2)$.

**Сложность** обхода **Джарвиса** - O(hn), где n - количество всех вершин и h - количество вершин попавших в выпуклую оболочку. **Алгоритм** обхода Грэхема **Сложность** данного **алгоритма** O(nlgn) - следовательно он будет работать быстрее обхода **Джарвиса**.

# Структура.

Алгоритм Джарвиса (другое название — алгоритм заворачивания подарков) концептуально устроен проще алгоритма Грэхема. Он является двухшаговым и

не требует сортировки. Первый шаг точно такой же — нам нужна стартовая точка, которая гарантированно входит в МВО, берем самую левую точку из A.

```
def jarvismarch(A):

  n = len(A)

  P = range(n)

  for i in range(1,n):

    if A[P[i]][0]<A[P[0]][0]:

      P[i], P[0] = P[0], P[i]
```

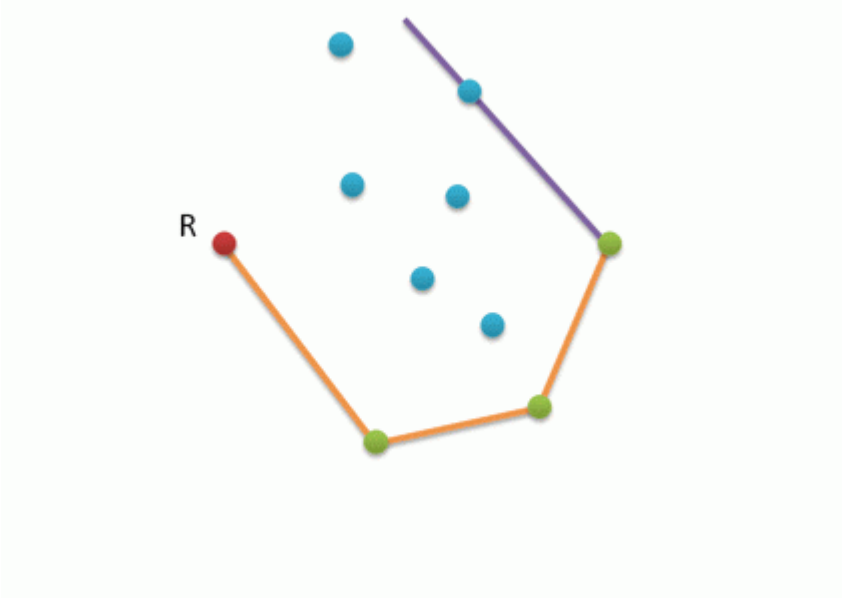
На втором шаге алгоритма строится МВО. Идея: делаем стартовую вершину текущей, ищем самую правую точку в A относительно текущей вершины, делаем ее текущей и т.д. Процесс завершается, когда текущей вновь окажется стартовая вершина. Как только точка попала в МВО, больше ее можно не учитывать. Поэтому заведем еще один список H, в котором в правильном порядке будут храниться вершины МВО. В этот список сразу же заносим стартовую вершину, а в списке P эту вершину переносим в конец (где мы ее в конце концов найдем и завершим алгоритм).

```
H = [P[0]]

  del P[0]

  P.append(H[0])
```

Теперь организуем бесконечный цикл, на каждой итерации которого ищем самую левую точку из P относительно последней вершины в H. Если эта вершина стартовая, то прерываем цикл, если нет — то переносим найденную вершину из P в H. После завершения цикла в H находится искомая оболочка, которую мы и возвращаем в качестве результата.

```
while True:
```

```
    right = 0
    for i in range(1,len(P)):
      if rotate(A[H[-1]],A[P[right]],A[P[i]])<0:
        right = i
    if P[right]==H[0]:
      break
    else:
      H.append(P[right])
      del P[right]
  return H
```




Оценим сложность алгоритма Джарвиса. Первый шаг линеен по n. Со вторым все интереснее. У нас имеется вложенный цикл, число внешних итераций равно числу вершин h в MBO, число внутренних итераций не превышает n. Следовательно, сложность всего алгоритма равна O(hn). Необычным в этой формуле является то, что сложность определяется не только длиной входных данных, но и длиной выхода (output-sensitive algorithm). А дальше как ~~карты~~ точки лягут. В худшем случае все точки из A входят в MBO (т.е. A уже само по себе выпуклый многоугольник), тогда h=n и сложность подскакивает до квадратичной. В лучшем случае (при условии, что точки из A не лежат на одной прямой) h=3 и сложность становится линейной. Осталось заранее

понять, какой у нас случай, что сделать не так просто (если у вас нет машины времени**), можно только исходить из характера задачи — если точек много и они равномерно заполняют некоторую область, то (возможно) Джарвис будет быстрее, если же данные собраны на границе области, то быстрее будет Грэхем.

## Полный код

```
#include <iostream>

#include <cstdio>

#include <vector>

#include <cmath>

 using namespace std;

 struct point {

    int x,y;

    point(){}

    point(int X, int Y)

    {

        x = X;

        y = Y;

    }

};

bool operator != (const point &a, const point &b)

 return !(a.x == b.x && a.y == b.y);
```

```
}
double dist (const point &a, const point &b)
{
    return sqrt( 0.0 + (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
int n;
vector<point> mas;
vector<int> convex_hull;
double P;
void input()
{
    cin>>n;
    mas.resize(n);
    for (int i=0;i<n;i++)
        scanf("%d %d", &mas[i].x, &mas[i].y);
}
int OrientTriangl2(const point &p1,const point &p2, const point &p3)
{
    return p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y) + p3.x * (p1.y - p2.y);
}

bool isInside(const point &p1, const point &p, const point &p2)
```

```
{
    return ( p1.x <= p.x && p.x <= p2.x &&
             p1.y <= p.y && p.y <= p2.y);
}
void ConvexHullJarvis(const vector<point> &mas, vector<int> &convex_hull)
{
    // находим самую левую из самых нижних
    int base = 0;
    for (int i=1;i<n;i++)
    {
        if (mas[i].y < mas[base].y)
            base = i;
        else
            if (mas[i].y == mas[base].y &&
                mas[i].x <  mas[base].x)
                base = i;
    }
    // эта точка точно входит в выпуклую оболочку
    convex_hull.push_back(base);
   int first = base;
    int cur = base;
    do
```

```
    {
        int next = (cur + 1) % n;
        for (int i=0;i<n;i++)
        {
            int sign = OrientTriangl2(mas[cur], mas[next], mas[i]);
            // точка mas[i] находится левее прямой ( mas[cur], mas[next] )
            if (sign < 0) // обход выпуклой оболочки против часовой стрелки
                next = i;
            // точка лежит на прямой, образованной точками  mas[cur], mas[next]
            else if (sign == 0)
            {
                // точка mas[i] находится дальше, чем mas[next] от точки mas[cur]
                if (isInside(mas[cur],mas[next],mas[i]))
                    next = i;
            }
        }
        cur = next;
        convex_hull.push_back(next);
    }
    while (cur != first);
}
void solve()
```

```
{
    ConvexHullJarvis(mas,convex_hull);
    for (size_t i=0;i<convex_hull.size()-1;i++)
        P += dist(mas[convex_hull[i]],mas[convex_hull[i+1]]);
}
void output()
{
    printf("%0.1f",P);
}
int main()
{
  input();
    solve();
    output();
   return 0;
}
```

## Только сам алгоритм Джарвиса

```
def jarvismarch(A):
  n = len(A)
  P = range(n)
```

```
 # start point
 for i in range(1,n):
   if A[P[i]][0]<A[P[0]][0]:
     P[i], P[0] = P[0], P[i]
 H = [P[0]]
 del P[0]
 P.append(H[0])
 while True:
   right = 0
   for i in range(1,len(P)):
     if rotate(A[H[-1]],A[P[right]],A[P[i]])<0:
       right = i
   if P[right]==H[0]:
     break
   else:
     H.append(P[right])
     del P[right]
 return H
```

## Доказательство корректности алгоритма

Выберем какую-нибудь точку $p_0$, которая гарантированно попадёт в выпуклую оболочку. Например, нижнюю, а если таких несколько, то самую левую из них.

Дальше будем действовать так: найдём самую «правую» точку от последней добавленной (то есть точку с минимальным полярным углом) и добавим её в оболочку. Будем так итеративно добавлять точки, пока не «замкнёмся», то есть пока самой правой точкой не станет $p_0$.

Корректность алгоритма легко доказывается по индукции:

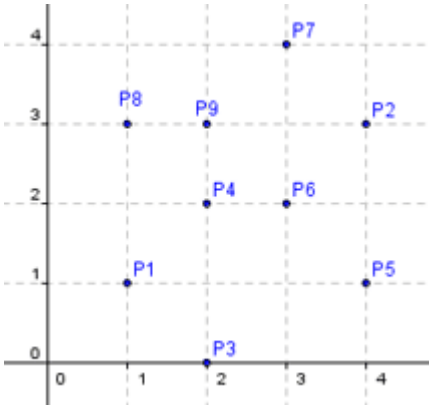
- На первом шаге мы выбрали точку, точно лежащую в МВО.
- На $i$-м шаге мы взяли такую точку, что все остальные лежат по «правильную» сторону отрезка $(p_i, p_{i+1})$.

Алгоритм Джарвиса также называют алгоритмом заворачивания подарка: мы каждый раз находим самый близкий «угол».

Для каждой точки выпуклой оболочки (обозначим их количество за $h$) мы из всех оставшихся $O(n)$ точек будем искать оптимальную, что суммарно будет работать за $O(n h)$.

Важно помнить, что асимптотика именно $O(nh)$, а не $O(n^2)$: существуют задачи, где оболочка маленькая, и это существенно.

## График работы алгоритма Джарвиса




расстановка точек

Входные данные:
9
1 1 4 3 2 0 2 2 4 1 3 2 3 4 1 3 2 3
Выходные данные:
2 0 4 1 4 3 3 4 1 3 1 1

# Литература


- *Прапарата Ф., Шеймос М.* Вычислительная геометрия: Введение = Computational Geometry An introduction. — М.: Мир, 1989. — С. 478.
- *Ласло М.* Вычислительная геометрия и компьютерная графика на C++. — М.: БИНОМ, 1997. — С. 304.
- *Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.* Алгоритмы. Построение и анализ = Introduction to Algorithms. — 2-е изд. — “Вильямс”, 2005. — С. 1296.


## Источники


1) https://habr.com/ru/post/144921/

2) https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B6%D0%B0%D1%80%D0%B2%D0%B8%D1%81%D0%B0

3) http://cppalgo.blogspot.com/2010/11/blog-post.html

4) https://dic.academic.ru/dic.nsf/ruwiki/614134

А это просто 3 D выпуклая оболочка....