Национална лагер-школа по информатика
Ямбол, 2024 г.

СЪСТЕЗАТЕЛНА ГЕОМЕТРИЯ
8 клас

Пламенка Христова

Вектори

- **Определение**

В математиката евклидов вектор (понякога наричан геометричен или пространствен вектор) или просто вектор е геометричен обект, който има величина (или дължина) и посока.

- **Всеки вектор вектор AB има следните елементи:**

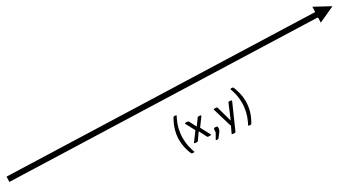
1) начало (приложна точка) A;
2) край B;
3) посока – посоката, в която се движи една точка, описваща вектора от началото A към края B;
4) директриса – правата AB, върху която лежи векторът вектор AB ($A \neq B$)
5) дължина (модул, големина) – дължината на отсечката AB при избрана единична отсечка, която бележим |вектор AB|

Вектори

- **Дължина на вектор**

Ако ни е даден вектор (x,y), дължината му е:

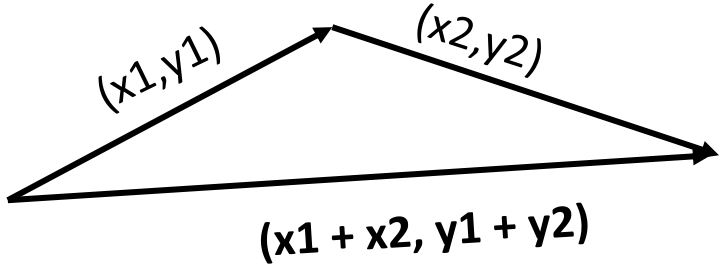
$$|(x,y)| = \sqrt{x^2 + y^2}$$



- **Събиране (изваждане) на вектори**

Ако са ни дадени два вектора (x1,y1) и (x2,y2), сумата (разликата) им е:

(x1, y1) +/- (x2, y2) = (x1 +/- x2, y1 +/- y2)



- **Скаларно произведение**

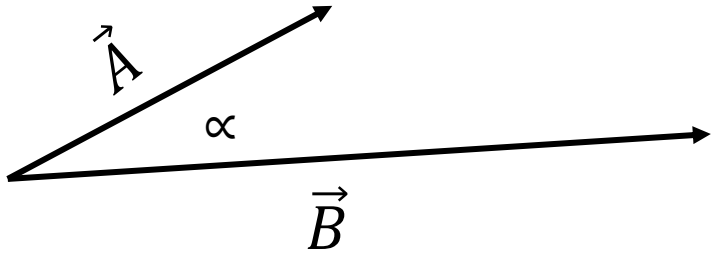
Скаларното произведение на 2 вектора е **число**, равно на сумата от призведенията на съответните елементи на векторите:

$$(x1, y1) . (x2, y2) = x1x2 + y1y2$$

където (x1, y1) и (x2, y2) са двата дадени вектора.

- Ако A и B са два вектора, а α е ъгъла между тях, то скаларното им произведение е:

$$A.B = \cos(\alpha)|A|.|B|$$



- **Векторно произведение**

Векторното призведение на 2 вектора е **вектор**, препендикулярен на равнината, образувана от 2-та вектора, който има дължина **|(x1*y2 - x2*y1)|** и посока, такава че трите вектора да са положително ориентирани в пространството.

Тъй като ние работим предимно в равнината можем да си милим за векторното произведение като число:

$$(x1, y1) \times (x2, y2) = x1*y2 - x2*y1$$

Нека A и B са два вектора, а α е ъгъла между тях, тогава

$$A \times B = \sin(\alpha)\ |A|.|B|$$

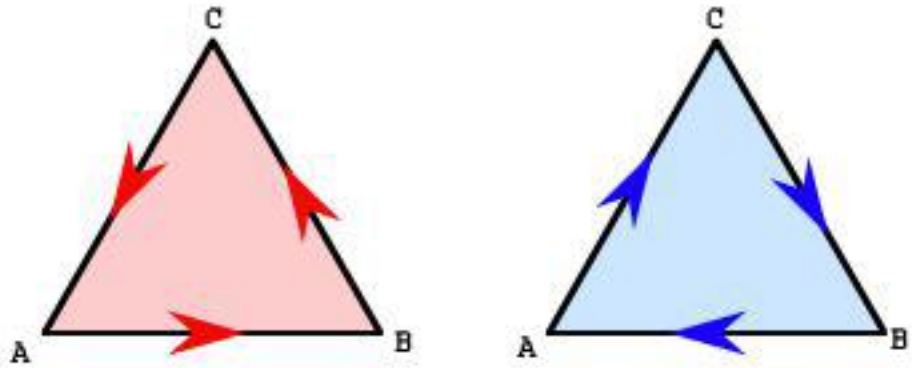
Знакът на α има значение:

Ако α е по- малко от 180 градуса A е по часовниковата стрелка спрямо B, тогава α е положително.

Ако образуваме триъгълник от двата вектора, тогава неговото лице е точно

$$S = |A \times B|/2$$

Ориентирано лице на триъгълник



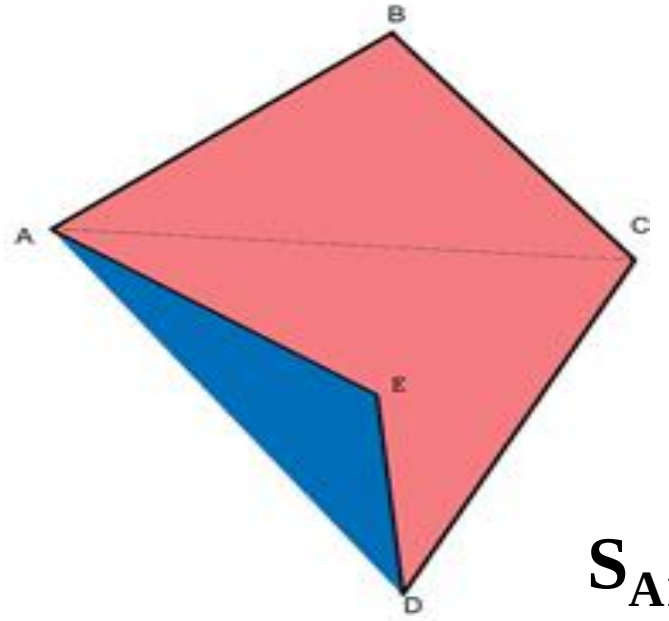
$$S_{ABC} = S_{BCA} = S_{CAB} = -S_{ACB} = -S_{CBA} = -S_{BAC}$$

Ориентирантото лице на тригълника ABC е положително число, ако точките A, B и C са наредени в **положителна (обратна на часовниковата стрелка) посока**. Тоест, ако вземем една вътрешна за триъгълника точка и започнем да въртим една стрелка през нея в положително посока ще посочваме точките A B C A B C A B C ...

Ако **лицето е отрицателно**, при същата процедура ще посочваме точките A C B A C B A C B...

$$\mathbf{S_{ABC} = (\overrightarrow{AB} \times \overrightarrow{AC})/2}$$

Ориентирано лице на многоъгълник



$$S_{A1,A2,\ldots,AN} = S_{A1,A2,A3} + S_{A1,A3,A4} + \cdots + S_{A1,AN-1,AN}$$

Формулата е вярна не само за изпъкнали многолъгълници, тъй като "вдлъбнатите" (сини на чертежа) части в един многоъгълник имат обратна ориентация спрямо "изпъкналите" (розови на чертежа).

Една примерна реализация:

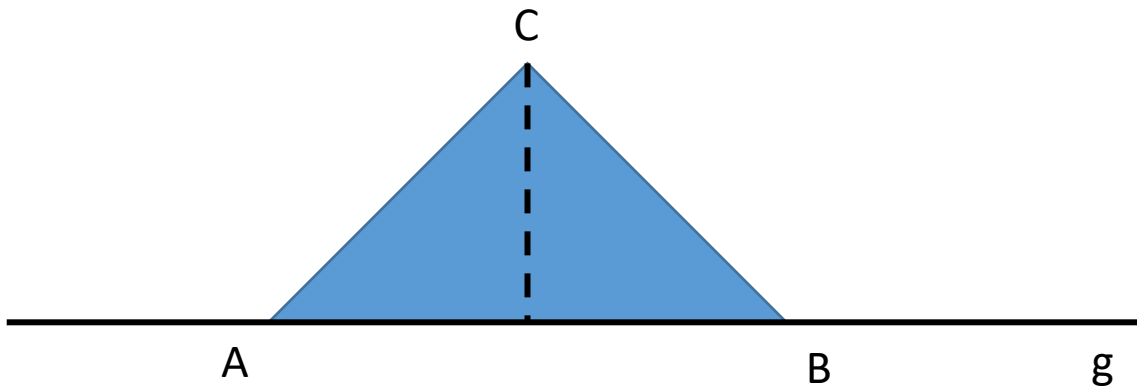
Приемаме, че предварително сме записали точките в двумерен масив A[N][2].

```
double Area()
{
  double S = 0;
  for(int i = 1; i < N-1; ++i)
  {
    double x1 = A[i][0] - A[0][0];
    double y1 = A[i][1] - A[0][1];
    double x2 = A[i+1][0] - A[0][0];
    double y2 = A[i+1][1] - A[0][1];
    S += x1*y2 - x2*y1;
  }
  return S/2;
}
```

Разстояние между права (отсечка) и точка

Нека ни е дадена правата g, определена от точките A и B, и искаме да намерим разстоянието d от g до точката C. Това разсояние е равно на височината на триъгълник ABC към страната AB, тоест на лицето на триъгълника, разделено на дължината на AB:

$$\mathbf{d = |(\overrightarrow{AB} \times \overrightarrow{AC})/2| \ / \ |AB|}$$



```
 // разсточние м/у A и B
double length(double[] A, double[] B) {
  double x = B[0] - A[0];
  double y = B[1] - A[1];
  return sqrt(x*x + y*y);
}
//векторно призведение AB x AC
double vector(double [] A, double [] B, double [] C) {
  double AB[2] = { B[0] - A[0], B[1] - A[1]};
  double AC[2] = { C[0] - A[0], C[1] - A[1]};
  return AB[0] * AC[1] - AB[1] * AC[0];
}
double lineDist(double[] A, double[] B, double[] C) {
  retrun abs( vector(A,B,C) / length(A,B) );  }
```

Разстояние от точка C до отсечка AB

Разглеждаме случая, когато петата на перпендикуляра от точка C към правата AB е извън отсечката AB. Тогава най-близката точка ще бъде или A, или B.

За да проверим това, изчисляваме скаларното призведение между $\overrightarrow{AB}$ и $\overrightarrow{BC}$. Ако то е > 0, тогава ъгълът между $\overrightarrow{AB}$ и $\overrightarrow{BC}$ е между -90 и 90 градуса (т.е. ъгъла ABC е тъп) и съответно точката B е най-близката точка от отсечката до C.

Същата проверка правим и за $\overrightarrow{BA}$ и $\overrightarrow{CA}$.

```
//скаларно призведение AB . BC
double skalar(double[] A, double[] B, double[] C)
{
  double AB[2] = { B[0] - A[0], B[1] - A[1]};
  double BC[2] = { B[0] - C[0], B[1] - C[1]};
  return AB[0] * BC[0] + AB[1] * BC[1];
}

double segmentDist(double[] A, double[] B, double[] C)
{
  if(skalar(A,B,C) > 0)
     return length(B,C);

  else if( skalar(B,A,C) > 0 )
     return length(A,C);

  else
     return abs( vector(A,B,C) / length(A,B) );
}
```

Да предположим, че имаме 2 прави, зададени с общи уравнения:

$$\mathbf{A_1*x + B_1*y + C_1 = 0}$$
$$\mathbf{A_2*x + B_2*y + C_2 = 0}$$

Искаме да намерим пресечната им точка, в случай че има такава. Двете уравнения образуват проста линейна система.
Ако детерминатата:

$$\begin{vmatrix} A1 & B1 \\ A2 & B2 \end{vmatrix} = A_1*B_2 - A_2*B_1 = 0$$

на системата има 0 или безброй много решения - правите са успоредни или съвпадат.

В противен случай решаваме системата, за да получим единствената пресечна точка (X, Y):

$$\mathbf{X = (B_1*C_2 - B_2*C_1)/(A_1*B_2 - A_2*B_1)}$$
$$\mathbf{Y = (A_2*C_1 - A_1*C_2)/(A_1*B_2 - A_2*B_1)}$$

Да разгледаме случая, в който пресичаме 2 отсечки.

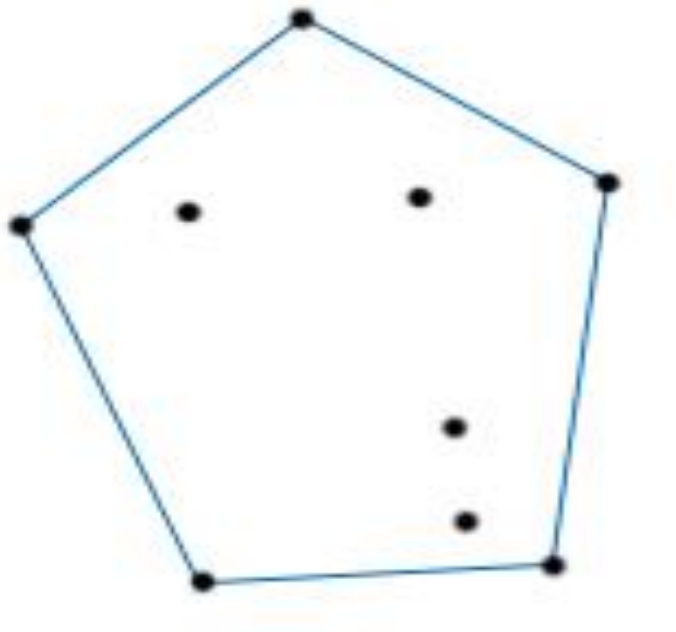
Трябва да проверим дали точката лежи и на двете отсечки едновременно.

Ако отсечката е с краища в точки с координати (x_1, y_1) и (x_2, y_2), а точката е с координати (x, y), то трябва да проверим дали са изпълнени неравенствата:

$$\mathbf{\min(x_1,x_2) \leq x \leq \max(x_1,x_2)}$$

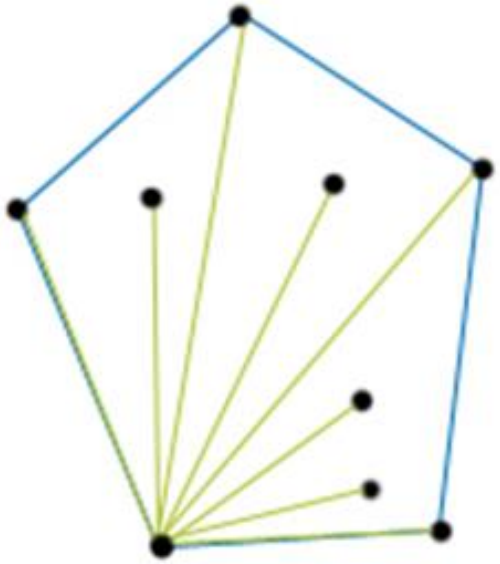
$$\mathbf{\min(y_1,y_2) \leq y \leq \max(y_1,y_2)}$$

Изпъкнала обвивка



Изпъкналата обвивка на множество S от точки е **единственият изпъкнал многоъгълник с върхове точки от S**, който съдържа всички тички от S.

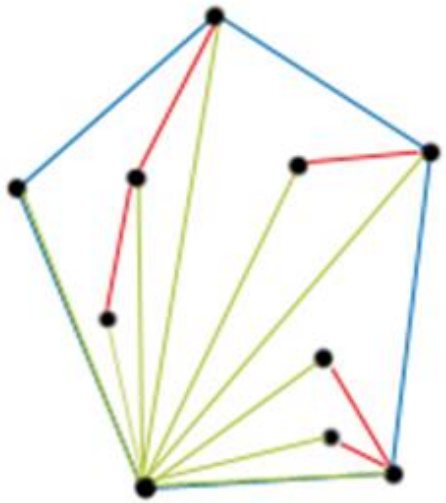
Алгоритъм на Graham за намиране на изпъкнала обвивка



В началото на алгоритъма вземаме празен стек T

1. Намираме най-доланата и най-ляво разположена точка от множеството. Нека я означим с P. Тя със сигурност принадлежи на изпъкналата обвивка
2. Сортираме останалите точки по големината на ъгъла, който сключва правата минаваща през съответната точка и точката P с абцисата. По този начин наредената последователност от сортираните точки и P образуват многоъгълник, в който всички точки са видими от P.

Алгоритъм на Graham за намиране на изпъкнала обвивка на многоъгълник



3. Добавяме в стека T точката P и първата точка от сортираната последователност.

Те със сигурност влизат в изпъкналата обвивка.

Червените линии показват, че вадим точка от стека.

4. Обхождаме последователно всички точки от сортираната последователност (без първата, защото вече е в стека) и за всяка повтаряме следното:

   Нека C е точката, която обхождаме, B е точката на върха на стека, а A е точката "под" върха на стека.

   Ако ъгълът ABC е по-малък от 180 (в положителна посока) вадим B от стека и отиваме към 4, в противен случай добавяме C в стека и преминаваме към следващата точка.

5. В стека T вече имаме изпъкнала обвивка на множеството.

Алгоритъм на Andrew за намиране на изпъкнала обвивка на множество от точки в равнината

1. Точките трябва да се сортират в нарастващ ред на x координатите им, ако те са свързани – и по y координатите им.
2. Построяваме горна и долна изпъкнала обвивки.

   Горна изпъкнала обвивка се нарича част от цялата изпъкнала обвивка, в която всички точки се виждат отгоре. Тя се движи от най-дясната си точка до най-лявата в посока обратна на часовниковата стрелка.

   Долната изпъкнала обвивка е останалата част от изпъкналата обвивка.

Алгоритъмът, зададен като псевдокод: O (n log n)

Вход: множество P от точки в равнината.

Сортираме точките от P по x-координатите (когато са свързани - по y-координатите).

Инициализираме два празни списъка U и L. В тях ще записваме върховете, които образуват горната и долна изпъкнала обвивка.

```
for i = 1, 2, ..., n:
```

докато L съдържа поне 2 точки и наредената тройка от последните 2 точки с точка P[i] е обратна на часовниковата стрелка:

изтрий последната точка от L

добави P[i] към L

Алгоритъмът, зададен като псевдокод:

```
for i = n, n-1, ..., 1:
```

докато U съдържа поне 2 точки и наредената тройка от последните 2 точки с точка P[i] е обратна на часовниковата стрелка:

изтрий последната точка от U

добави P[i] към U

Изтрий последната точка от всеки списък (тя е първа от другия).
Свържи L и U, за да получиш изпъкналата обвивка на P.

Точките в резултата ще се изведат в посока, обратна на часовниковата стрелка.

```
// Implementation of Andrew's monotone chain 2D convex hull
algorithm.
// Asymptotic complexity: O(n log n).
#include <algorithm>
#include <vector>
using namespace std;

typedef double coord_t;         // coordinate type
typedef double coord2_t;  // must be big enough to hold
2*max(|coordinate|)^2

struct Point {
     coord_t x, y;
     bool operator <(const Point &p) const {
          return x < p.x || (x == p.x && y < p.y);
     }
};
```

```
// Връща положителна стойност, ако OAB е обратно на
// часовниковата стрелка, отрицателна стойност по посока на
часовниковата стрелка, и 0 ако точките са колинеарни.

coord2_t cross(const Point &O, const Point &A, const Point &B)
{
 return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Връща списък от точки на изпъкналата обвивка, разположени
// по посока на часовниковата стрелка.

vector<Point> convex_hull(vector<Point> P)
{
     int n = P.size(), k = 0;
     if (n == 1) return P;
     vector<Point> H(2*n);
```

```
    // Сортиране на точките
    sort(P.begin(), P.end());

    // Построяване на долна изпъкнала обвивка
    for (int i = 0; i < n; ++i) {
       while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
         H[k++] = P[i];
    }

    // Построяване на горна изпъкнала обвивка
    for (int i = n-2, t = k+1; i >= 0; i--) {
       while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
         H[k++] = P[i];
    }
    H.resize(k-1);
    return H;
}
```

Пресичане на две прави (отсечки)

Общо уравнение на права:

$$Ax + By + C = 0$$

В повечето задачи правата е зададена с две точки P и Q исе налага сами да пресметнем стойностите на коефициентите A, B и C.

Ето как става това за точките $P(x_1, y_1)$ и $Q(x_2, y_2)$:

$$A = y_1 - y_2$$

$$B = x_2 - x_1$$

$$C = x_1y_2 - x_2y_1$$

Както и да са дадени правите, винаги има начин да ги представим посредством общото им уравнение.

Уравнение на окръжност по зададени три точки на окръжността

Общият случай на канонично уравнение на окръжност с център в точката O(h, k) и радиус r е: $(x-h)^2+(y-k)^2=r^2$

Да се намери уравнението на окръжността k през точките A(2, −2), B(7, 3), D(5, −1).
Един начин да решим задачата е да построим ***два диаметъра на търсената окръжност*** и намирайки пресечната им точка, да получим координатите на центъра на k.
Нека M и N са среди съответно на хордите AB и BD. Тогава M (9/2 , 1/2) и N (6, 1).
Построяваме диаметъра d1 като права през M и перпендикулярна на AB.
Аналогично, построяваме диаметъра d2 като права през N и перпендикулярна на BD.
Така получаваме d1 : x + y − 5 = 0, d2 : x + 2y − 8 = 0.
Пресечната точка на тези две прави C има координати C(2, 3) и е център на търсената окръжност. Радиусът на окръжността е $r = |\overrightarrow{CA}| = |\overrightarrow{CB}| = |\overrightarrow{CD}| = 5$.
Следователно търсената окръжност има уравнение k : $(x − 2)^2 + (y − 3)^2 = 25$

Минимална обхващаща окръжност

Даден е масив arr[][], съдържащ N точки в 2-D равнина с целочислени координати. Задачата е да се намерят центърът и радиусът на минималната обхващаща окръжност (MEC).

Минималната обхващаща окръжност е окръжност, в която всички точки лежат или вътре в окръжността, или по границите.

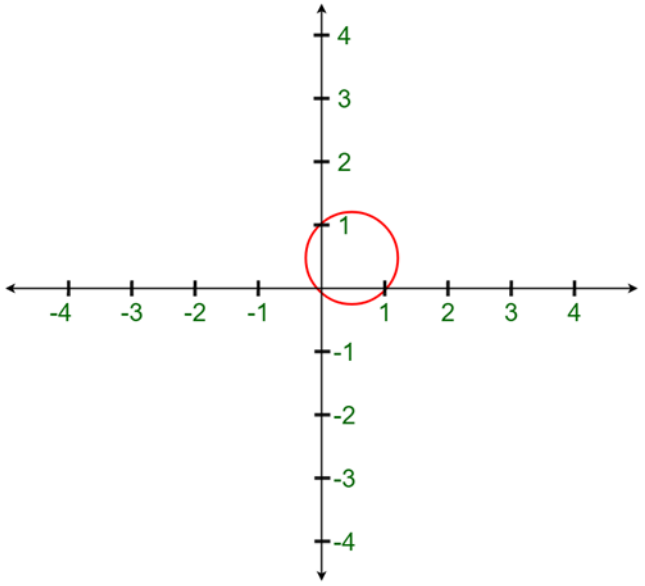
Пример:

Вход: arr[][] = {{0, 0}, {0, 1}, {1, 0}}

Изход: център = {0,5, 0,5}, радиус = 0,7071

Обяснение:

При начертаване на горната окръжност с радиус 0,707 и център (0,5, 0,5), може ясно да се види, че и трите точки лежат или вътре, или върху окръжността.



Алгоритъм на Welzl за определяне на минимална обхващаща окръжност (MEC) на облак от точки в равнината

Наивен подход:

Първо намираме множеството точки, които определят изпъкналата обвивка на множеството от точки и след това построяваме всички възможни окръжности по дадени три точки и за всяка от тях проверяваме дали останалите точки от изпъкналата обвивка влизат в окръжността. Най-лошият случай на времева сложност след тази оптимизация е $O(N^4)$.

Алгоритъм на Welzl за определяне на минимална обхващаща окръжност (MEC) на облак от точки в равнината

Използвайки рекурсивният алгоритъм на Welzl тази MEC може да бъде намерена за O(N).

Идеята на алгоритъма е произволно да премахне точка от дадения входен набор от точки, за да формира уравнение на окръжност. След като уравнението е съставено, се проверява дали премахнатата точка е в окръжността или не. Ако не е, тогава точката трябва да лежи на границата на MEC. Следователно тази точка се счита за гранична точка и функцията се извиква рекурсивно.

Алгоритъм на Welzl

Алгоритъмът използва две множества: множеството от точки P и множество R, което първоначално е празно и се използва за представяне на точките на границата на MEC като вход.

Базовият случай на алгоритъма е, когато P стане празно или размерът на R е равен на 3: Ако P е празно, всички точки са обработени. Ако |R| = 3, тогава вече са намерени 3 точки, които лежат на границата на окръжността, и тъй като окръжност може да бъде еднозначно определена само с 3 точки, рекурсията може да бъде спряна.

Алгоритъм на Welzl

Когато алгоритъмът достигне ***основния случай*** по-горе, той връща тривиалното решение за R, което е:

- Ако |R| = 1, връщаме кръг с център R[0] с радиус = 0
- Ако |R| = 2, връщаме MEC за R[0] и R[2]
- Ако |R| = 3, връщаме MEC като пробваме 3-те двойки (R[0], R[1]), (R[0], R[2]) и (R[1], R[2]). Ако нито една от тези двойки не е валидна, връщаме окръжността, дефинирана от 3-те точки в R

Ако базовият случай все още не е достигнат, правим следното: Избираме произволна точка p от P и я премахваме от P. Извикваме алгоритъма за P и R, за да получим кръг d. Ако p е в d, тогава връщаме d, в противен случай p трябва да лежи на границата на MEC. Добавяме p към R. Връщаме изхода на алгоритъма за P и R.

```
#include <algorithm>
#include <assert.h>
#include <iostream>
#include <math.h>
#include <vector>
using namespace std;

const double INF = 1e18; // Безкрайност

struct Point {
	double X, Y;
};
struct Circle {
	Point C;
	double R;
};
```

```
// минимална обхващаща окръжност  при N <= 3
Circle min_circle_trivial(vector<Point>& P){
	assert(P.size() <= 3);
	if (P.empty()) {
		return { { 0, 0 }, 0 };
	}
	else if (P.size() == 1) {
		return { P[0], 0 };
	}
	else if (P.size() == 2) {
		return circle_from(P[0], P[1]);
	}
// проверка дали MEC може да се определи само с 2 точки
	for (int i = 0; i < 3; i++)
		for (int j = i + 1; j < 3; j++) {
			Circle c = circle_from(P[i], P[j]);
			if (is_valid_circle(c, P))return c;
		}
	return circle_from(P[0], P[1], P[2]);
}
```

```
// Връща MEC, използвайки алгоритъма на Welzl.
Circle welzl_helper(vector<Point>& P, vector<Point> R, int n){
 // Базов случай, когато всички точки са обработени или |R| = 3
      if (n == 0 || R.size() == 3) return min_circle_trivial(R);
 // Избор на случайна точка
      int idx = rand() % n;
      Point p = P[idx];
 // Поставяне избраната точка в края на P,
 // тъй като е по-ефективно от изтриването от средата на вектора
      swap(P[idx], P[n - 1]);
 // Получаване MEC окръжността d от множеството от точки P - {p}
      Circle d = welzl_helper(P, R, n - 1);
 // Ако d съдържа p, връщаме d
      if (is_inside(d, p))return d;
// В противен случай, трябва да лежи на контура на MEC
      R.push_back(p);
// Return the MEC for P - {p} and R U {p}
      return welzl_helper(P, R, n - 1);
}
```

```
// Otherwise, must be on the boundary of the MEC
      R.push_back(p);
// Return the MEC for P - {p} and R U {p}
      return welzl_helper(P, R, n - 1);
}
Circle welzl(const vector<Point>& P){
      vector<Point> P_copy = P;
      random_shuffle(P_copy.begin(), P_copy.end());
      return welzl_helper(P_copy, {}, P_copy.size());
}
int main(){
      Circle mec = welzl({ { 0, 0 },{ 0, 1 },{ 1, 0 } });
      cout << "Center = { " << mec.C.X << ", " << mec.C.Y
            << " } Radius = " << mec.R << endl;

      Circle mec2 = welzl({ { 5, -2 },{ -3, -2 },{ -2, 5 },{ 1, 6 },
                                    { 0, 2 } });
      cout << "Center = { " << mec2.C.X << ", " << mec2.C.Y
            << " } Radius = " << mec2.R << endl;
      return 0;
}
```

ПТИ, 2016 г., Задача С3. ФИГУРИ
ЗТИ, 2009 г., Задача С2. ДОЛИНИ
ЕТИ, 2011 г., Задача С1. ТРИЪГЪЛНИЦИ