

## Geometrijski algoritmi

Geometrijski algoritmi igraju važnu ulogu u mnogim oblastima, poput računarske grafike, projektovanja pomoću računara, projektovanju integrisanih kola visoke rezolucije (VLSI), robotike i baza podataka. U računarski generisanoj slici može biti na hiljade ili čak na milione tačaka, linija, kvadrata ili krugova; slično, projektovanje kompjuterskog čipa može da zahteva rad sa milionima elemenata. Svi ovi zahtevaju obradu geometrijskih objekata. Pošto veličina ulaza za ove probleme može biti vrlo velika, veoma je značajno razviti što efikasnije algoritme za njihovo rešavanje.

U nastavku ćemo razmotriti nekoliko osnovnih geometrijskih algoritama — onih koji se mogu koristiti kao elementi za izgradnju složenijih algoritama. Objekti sa kojima se radi su tačke, prave, duži i mnogouglovi. Tačka  $P$  u ravni predstavlja se parom koordinata  $(x, y)$ . Prava je predstavljena parom različitih tačaka  $P$  i  $Q$  koje joj pripadaju. Duž se takođe zadaje parom tačaka  $P$  i  $Q$  koje predstavljaju krajeve te duži, i označavaćemo je sa  $PQ$ . Put  $P$  je niz tačaka  $P_1, P_2, \dots, P_k$  i duži  $P_1P_2, P_2P_3, \dots, P_{k-1}P_k$  koje ih povezuju. Duži koje čine put su njegove stranice (ivice). Zatvoreni put je put čija se poslednja tačka poklapa sa prvom i nazivamo ga mnogougao ili poligon. Tačke koje definišu mnogougao su njegova temena. Na primer, trougao je mnogougao sa tri temena. Mnogougao se predstavlja nizom, a ne skupom tačaka, jer je bitan redosled kojim se tačke zadaju; promenom redosleda tačaka iz istog skupa u opštem slučaju dobija se drugi mnogougao. Prost mnogougao je onaj kod koga odgovarajući put nema preseka sa samim sobom; drugim rečima, jedine stranice koje imaju zajedničke tačke su susedne stranice sa svojim zajedničkim temenom. Prost mnogougao ograničava jednu oblast u ravni, unutrašnjost mnogougla. Konveksni mnogougao je mnogougao čija unutrašnjost sa svake dve tačke koje sadrži, sadrži i sve tačke duži koje te tačke određuju (ekvivalentno, mnogougao je konveksan ako su mu svi unutrašnji uglovi manji ili jednaki  $180^\circ$ ). Konveksni put je put sastavljen od tačaka  $P_1, P_2, \dots, P_k$  takav da je mnogougao  $P_1P_2 \dots P_k$  konveksan.

Česta neugodna karakteristika geometrijskih problema je postojanje mnogih specijalnih slučajeva. Na primer, dve prave u ravni obično se seku u jednoj tački, sem ako su paralelne ili se poklapaju. Pri rešavanju problema sa dve prave, moraju se predvideti sva tri moguća slučaja. Složeniji objekti prouzrokuju pojavu mnogo većeg broja specijalnih slučajeva, o kojima treba voditi računa. Obično se većina tih specijalnih slučajeva neposredno rešava, ali potreba da se oni uzmu u obzir čini ponekad konstrukciju geometrijskih algoritama vrlo iscrpljujućom. Mi ćemo ponekad ignorisati detalje koji nisu od suštinskog značaja za razumevanje osnovnih ideja algoritama, ali čitaocu savetujemo da razmotri rešavanje svakog od specijalnih slučajeva na koji se pri konstrukciji algoritma naide.

## Osnovni algoritmi

Pretpostavlja se da je čitalac upoznat sa osnovama analitičke geometrije. U nastavku ćemo razmotriti neka osnovna pitanja na koja se nailazi prilikom konstrukcije gotovo svakog geometrijskog algoritma, kao što je izračunavanje rastojanja između dve tačke, ispitivanje kolinearnosti tri tačke ili izračunavanje presečne tačke dveju duži. Sve ove operacije mogu se izvesti u konstantnoj vremenskoj složenosti, korišćenjem osnovnih aritmetičkih operacija. Pri tome, na primer, pretpostavljamo da se kvadratni koren broja može izračunati za konstantno vreme.

### Skalarni i vektorski proizvod

Data su dva vektora  $\vec{a}(a_1, \dots, a_n)$  i  $\vec{b}(b_1, \dots, b_n)$  iste dimenzije. Njihov *skalarni proizvod* je skalar čiju vrednost računamo po formuli:

$$\vec{a} \circ \vec{b} = \sum_{i=1}^n a_i \cdot b_i$$

Ukoliko su koordinate vektora  $\vec{a}$  i  $\vec{b}$  celobrojne, i vrednost skalarnog proizvoda biće celobrojna.

```
// funkcija koja racuna skalarni proizvod dva vektora
int skalarniProizvod(vector<int> a, vector<int> b){
    // racunamo dimenziju vektora
    int n = a.size();
    // racunamo skalarni proizvod
    int proizvod = 0;
    for (int i = 0; i < n; i++)
        proizvod += a[i] * b[i];
    return proizvod;
}

int main(){
    vector<int> a = {2,2};
    vector<int> b = {3,4};
    cout << "Skalarni proizvod vektora a i b je: "
         << skalarniProizvod(a,b) << endl;
    return 0;
}
```

U daljem tekstu ćemo razmatrati vektore određene tačkama u ravni, te će dimenzija vektora biti jednaka 2. Skalarni proizvod vektora  $\vec{a}(a_x, a_y)$  i vektora  $\vec{b}(b_x, b_y)$  dimenzije 2 jednak je  $\vec{a} \circ \vec{b} = a_x \cdot b_x + a_y \cdot b_y$ .

Važi naredna formula:

$$\vec{a} \circ \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \cos \alpha$$

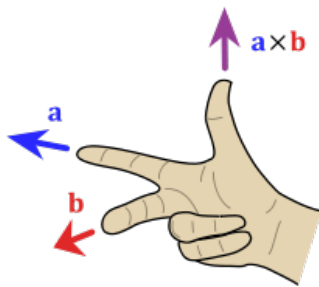
gde je sa  $|\vec{a}|$  označen intenzitet vektora  $\vec{a}$ , a sa  $\alpha$  ugao koji zaklapaju ova dva vektora. Na osnovu ove formule moguće je izračunati:

- intenzitet, tj. dužinu vektora  $\vec{u}$ , na osnovu formule  $|\vec{u}|^2 = \vec{u} \circ \vec{u}$ , jer je  $\cos 0^\circ = 1$ .
- ugao  $\alpha$  između vektora  $\vec{u}$  i  $\vec{v}$ , na osnovu formule  $\alpha = \arccos \frac{\vec{u} \circ \vec{v}}{|\vec{u}| \cdot |\vec{v}|}$

Vektorski proizvod vektora  $\vec{a}(a_x, a_y, a_z)$  i  $\vec{b}(b_x, b_y, b_z)$  dimenzije 3 je vektor ortogonalan na ravan određenu vektorima  $\vec{a}$  i  $\vec{b}$ , čiji je smer određen pravilom desne ruke (slika 1), a intenzitet jednak površini paralelograma koji određuju vektori  $\vec{a}$  i  $\vec{b}$ , odnosno može se izračunati po formuli:

$$|\vec{a} \times \vec{b}| = |\vec{a}| \cdot |\vec{b}| \cdot \sin \alpha$$

gde je sa  $\alpha$  označen manji od uglova koji obrazuju vektori  $\vec{a}$  i  $\vec{b}$ .



Slika 1: Pravilo desne ruke: ako kažiprst i srednji prst pokazuju redom u smeru vektora  $\vec{a}$  i  $\vec{b}$ , palac će određivati smer njihovog vektorskog proizvoda  $\vec{a} \times \vec{b}$ .

Neka je:

$$\begin{aligned}\vec{a} &= a_x \cdot \vec{i} + a_y \cdot \vec{j} + a_z \cdot \vec{k} \\ \vec{b} &= b_x \cdot \vec{i} + b_y \cdot \vec{j} + b_z \cdot \vec{k}\end{aligned}$$

gde su sa  $\vec{i}$ ,  $\vec{j}$  i  $\vec{k}$  označeni jedinični vektori u smeru  $x$ ,  $y$  i  $z$  koordinatne ose. Važi:

$$\begin{aligned}\vec{i} \times \vec{j} &= \vec{k} = -\vec{j} \times \vec{i} \\ \vec{j} \times \vec{k} &= \vec{i} = -\vec{k} \times \vec{j} \\ \vec{k} \times \vec{i} &= \vec{j} = -\vec{i} \times \vec{k}\end{aligned}$$

odakle dobijamo da je vektorski proizvod vektora  $\vec{a}$  i  $\vec{b}$  jednak:

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y) \vec{i} + (a_z b_x - a_x b_z) \vec{j} + (a_x b_y - a_y b_x) \vec{k}$$

Ova formula se može zapisati u obliku naredne determinante:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

u čijoj se prvoj vrsti nalaze jedinični vektori u smeru  $x$ ,  $y$  i  $z$  ose, u drugoj vrsti koordinate vektora  $\vec{a}$ , a u trećoj koordinate vektora  $\vec{b}$ . Ukoliko su koordinate vektora  $\vec{a}$  i  $\vec{b}$  celobrojne, i koordinate vektora  $\vec{a} \times \vec{b}$  biće celobrojne.

```
// funkcija koja racuna vektorski proizvod p vektora a i b
// u trodimenzionom prostoru
void vektorskiProizvod(vector<int> a, vector<int> b, vector<int>& p){
    p[0] = a[1]*b[2] - a[2]*b[1];
    p[1] = a[2]*b[0] - a[0]*b[2];
    p[2] = a[0]*b[1] - a[1]*b[0];
}

int main(){
    vector<int> a = {3,3,-4};
    vector<int> b = {1,-2,5};
    vector<int> proizvod(3);
    vektorskiProizvod(a,b,proizvod);
    cout << "Vektorski proizvod vektora a i b je: (" ;
    for(int i = 0; i < 3; i++)
        cout << proizvod[i] << " ";
    cout << ")" << endl;
    return 0;
}
```

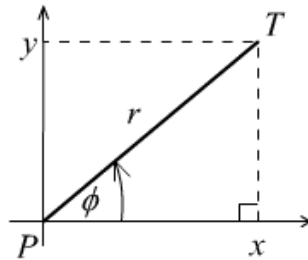
## Polarne koordinate

Nekada je za date Dekartove koordinate  $x$  i  $y$  tačke  $T$  potrebno odrediti njene polarne koordinate: rastojanje  $r$  od koordinatnog početka  $P$  i ugao  $\phi$  koji vektor  $\vec{PT}$  zahvata sa pozitivnim delom  $x$  ose i, obratno, na osnovu polarnih koordinata odrediti Dekartove.

Da bismo transformisali polarne koordinate u Dekartove, možemo iskoristiti sledeću vezu (slika 2):

$$\begin{aligned} x &= r \cos \phi \\ y &= r \sin \phi \end{aligned}$$

A za transformaciju Dekartovih koordinata u polarne, možemo iskoristiti naredne jednačine:



Slika 2: Veza između Dekartovih i polarnih koordinata tačke.

$$r = \sqrt{x^2 + y^2}$$

$$\phi = \arctan \frac{y}{x}$$

```
// transformacija Dekartovih koordinata u polarne
void dekartoveUPolarne(double x, double y, double &r, double &ugao){
    r = sqrt((pow(x,2)) + (pow(y,2)));
    ugao = atan2(y,x);
}

// transformacija polarnih u dekartove koordinate
void polarneUDekartove(double r, double ugao, double &x, double &y){
    x = r * cos(ugao);
    y = r * sin(ugao);
}

int main(){
    double x, y, r, ugao;
    cout << "Unesi Dekartove koordinate tacke" << endl;
    cin >> x >> y;
    // transformisemo Dekartove koordinate polazne tacke
    // u polarne koordinate
    dekartoveUPolarne(x,y,r,ugao);
    cout << "Polarne koordinate te tacke su r: "
        << r << " ugao: " << ugao << endl;

    double x1, y1;
    // transformisemo dobijene polarne koordinate u Dekartove
    // proveravamo da li smo dobili polazne koordinate
    polarneUDekartove(r,ugao,x1,y1);
    cout << "Dekartove koordinate te tacke su x:"
        << x1 << " y: " << y1 << endl;
    return 0;
}
```

```
}
```

## Površina trougla

**Problem:** Dat je trougao koordinatama svojih temena. Izračunati njegovu površinu.

Zadatak je matematički moguće rešiti na nekoliko načina. Jedan način je da primenimo Heronov obrazac koji kaže da je

$$P = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}$$

gde su sa  $a$ ,  $b$  i  $c$  označene dužine stranica trougla, a sa  $s = (a + b + c)/2$  njegov poluobim. Dužinu stranice trougla možemo izračunati kao rastojanje između njena dva temena.

Tačku ćemo nadalje predstavljati strukturom koja sadrži dve komponente:  $x$  i  $y$  koordinatu koje su realni brojevi. Alternativno, tačku bismo mogli da predstavimo uređenim parom dva realna broja `pair<double, double>`.

```
// tacku u ravni predstavljamo njenom x i y koordinatom
```

```
struct Tacka{  
    double x;  
    double y;  
};
```

```
// rastojanje izmedju tacaka
```

```
double rastojanje(Tacka A, Tacka B) {  
    double dx = B.x - A.x;  
    double dy = B.y - A.y;  
    return sqrt(dx * dx + dy * dy);  
}
```

```
// površina trougla za zadate dužine stranica,
```

```
// izracunata koriscenjem Heronovog obrasca
```

```
double površinaTrougla(double a, double b, double c) {  
    double s = (a + b + c) / 2;  
    return sqrt(s * (s - a) * (s - b) * (s - c));  
}
```

```
int main() {
```

```
    Tacka A, B, C;
```

```
    cout << "Unesi koordinate tacaka A, B i C" << endl;
```

```
    cin >> A.x >> A.y >> B.x >> B.y >> C.x >> C.y;
```

```
// racunamo dužine stranica trougla
```

```
    double a = rastojanje(B, C);
```

```

double b = rastojanje(A, C);
double c = rastojanje(A, B);

// racunamo površinu trougla
double P = površinaTrougla(a, b, c);
cout << "Površina trougla je: " << P << endl;
return 0;
}

```

Drugi način da dođemo do površine trougla  $ABC$  jeste korišćenjem činjenice da je intenzitet vektorskog proizvoda vektora  $\vec{AB}$  i  $\vec{AC}$  jednak površini paralelograma koji oni obrazuju. Pošto je naš trougao upravo polovina tog paralelograma, njegova površina jednaka je polovini intenziteta vektorskog proizvoda. Koordinate  $x$  i  $y$  vektora  $\vec{AB}$  i  $\vec{AC}$  moguće je izračunati oduzimanjem koordinata tačke  $A$  od koordinata tačaka  $B$  i  $C$ . Možemo pretpostaviti da naš trougao leži u ravni  $xOy$ , te je  $z$  koordinata ovih vektora jednaka 0. Vektorski proizvod dva vektora čije su koordinate poznate jednak je vrednosti naredne determinante:

$$\begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x) \quad (1)$$

Jedan način da se implementira površina trougla preko vektorskog proizvoda jeste da se iskoristi funkcija koja izračunava vektorski proizvod proizvoljna dva vektora dimenzije 3 i da se zatim izračuna njegov intenzitet.

```

// dužina (intenzitet) vektora u
double dužinaVektora(vector<double> u) {
    return sqrt(u[0] * u[0] + u[1] * u[1] + u[2] * u[2]);
}

// funkcija koja racuna vektorski proizvod p vektora a i b
// u trodimenzionom prostoru
void vektorskiProizvod(vector<double> a, vector<double> b,
    vector<double>& p){
    p[0] = a[1] * b[2] - a[2] * b[1];
    p[1] = a[0] * b[2] - a[2] * b[0];
    p[2] = a[0] * b[1] - a[1] * b[0];
}

// površina trougla zadatog temenima A, B i C
double površinaTrougla(Tacka A, Tacka B, Tacka C) {
    // vektor AB
    vector<double> AB;
    AB.push_back(B.x - A.x);
    AB.push_back(B.y - A.y);
}

```

```

AB.push_back(0);
// vektor AC
vector<double> AC;
AC.push_back(C.x - A.x);
AC.push_back(C.y - A.y);
AC.push_back(0);
// vektorski proizvod v = AB x AC
vector<double> v(3);
vektorskiProizvod(AB, AC, v);
// površina je polovina intenziteta vektorskog proizvoda
return duzinaVektora(v) / 2.0;
}

int main() {
    Tacka A, B, C;
    cout << "Unesi koordinate tacaka A, B i C" << endl;
    cin >> A.x >> A.y >> B.x >> B.y >> C.x >> C.y;

    // površina trougla
    double P = površinaTrougla(A, B, C);
    cout << "Površina trougla je: " << P << endl;
    return 0;
}

```

Pošto je u ovoj situaciji potrebno izračunati vektorski proizvod dva vektora koji leže u ravni  $xOy$ , tj. dva vektora kojima su  $z$  koordinate jednake nuli, izračunavanje je moguće optimizovati. Naime, rezultat će biti vektor normalan na ravan  $xOy$ , tj. koordinate  $x$  i  $y$  vektora biće jednake nuli. Intenzitet ovog vektora biće jednak apsolutnoj vrednosti koordinate  $z$  koja je po formuli (1) jednaka  $x_1y_2 - y_1x_2$ , pri čemu su  $(x_1, y_1)$  koordinate vektora  $\overrightarrow{AB}$ , a  $(x_2, y_2)$  koordinate vektora  $\overrightarrow{AC}$ . Dakle, površina trougla biće jednaka

$$\frac{|(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)|}{2}$$

tj.

$$\frac{|b_x c_y - a_x c_y - b_x a_y - b_y c_x + a_y c_x + a_x b_y|}{2}$$

```

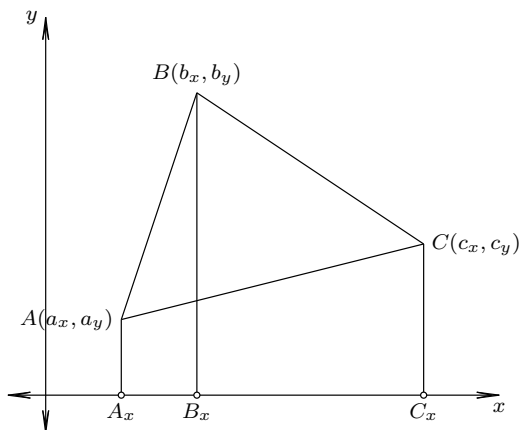
double površinaTrougla(Tacka A, Tacka B, Tacka C) {
    return abs(B.x*C.y - A.x*C.y - B.x*A.y
        - B.y*C.x + A.y*C.x + A.x*B.y)/2.0;
}

```

Napomenimo da je rešenje zasnovano na vektorskom proizvodu mnogo bolje nego ono zasnovano na Heronovom obrascu, jer koristi samo osnovne aritmetičke operacije (a ne i korenovanje).



Treći način dolaska do rešenja je da se površina trougla iskaže kao razlika površina određenih trapeza. Na primer, ako važi raspored  $a_x < b_x < c_x$ , i ako su  $A_x$ ,  $B_x$  i  $C_x$  redom projekcije tačaka  $A$ ,  $B$  i  $C$  na  $x$ -osu, tada je površina trougla  $ABC$  jednaka razlici između zbira površina pravouglavih trapeza  $AA_xB_xB$ ,  $BB_xC_xC$  i površine pravouglavih trapeza  $AA_xC_xC$  (slika 3).



Slika 3: Površina trougla izražena preko površina trapeza.

Površina trapeza  $AA_xB_xB$  jednaka je proizvodu dužine njegove srednje linije i dužine njegove visine. Dužina osnovice  $A_xA$  jednaka je apsolutnoj vrednosti koordinate  $a_y$ , dužina osnovice  $B_xB$  jednaka je apsolutnoj vrednosti koordinate  $b_y$  dok je dužina visine jednaka apsolutnoj vrednosti razlike koordinata  $b_x$  i  $a_x$ . Tako da je površina tog trapeza jednaka

$$\frac{|b_x - a_x|(|a_y| + |b_y|)}{2} \quad (2)$$

Na sličan način mogu se izvesti i formule za druga dva trapeza.

Analizom mogućih rasporeda tačaka, može se pokazati da će se ispravan rezultat dobiti i ako se posmatra takozvana označena površina trapeza, koja isključuje računanje apsolutne vrednosti u formuli (2) dopušta negativne dužine i površine. Označena površina trougla  $ABC$  biće jednaka zbiru označenih površina tri pomenuta pravouglavi trapeza ( $AA_xB_xB$ ,  $BB_xC_xC$  i  $CC_xA_xA$ ), a prava površina trougla biće jednaka njenoj apsolutnoj vrednosti. Označene površine ovih trapeza dobijaju se tako što se u prethodno izvedenim formulama za njihovu površinu zanemare apsolutne vrednosti i jednake su  $\frac{(b_x - a_x)(b_y + a_y)}{2}$ ,  $\frac{(c_x - b_x)(c_y + b_y)}{2}$  i  $\frac{(a_x - c_x)(a_y + c_y)}{2}$ , tako da se površina trougla dobija formulom

$$\frac{|(b_x - a_x)(b_y + a_y) + (c_x - b_x)(c_y + b_y) + (a_x - c_x)(a_y + c_y)|}{2} \quad (3)$$

```

// funkcija racuna oznacenu površinu pravouglog trapeza
// koji odredjuju tacke M, N i njihove projekcije na x-osu
double površinaTrapeza(Tacka M, Tacka N) {
    return (N.x - M.x) * (M.y + N.y);
}

// površina trougla zadatog temenima A, B, C
double površinaTrougla(Tacka A, Tacka B, Tacka C) {
    return abs(površinaTrapeza(A, B) +
               površinaTrapeza(B, C) +
               površinaTrapeza(C, A)) / 2.0;
}

```

Izraz iz formule (3) možemo srediti i dobijamo da je površina trougla jednaka:

$$\frac{|b_x a_y - a_x b_y + c_x b_y - b_x c_y + a_x c_y - c_x a_y|}{2}$$

Primetimo da smo na ovaj način dobili potpuno istu formulu kao kada smo površinu trougla računali pomoću vektorskog proizvoda. Ova formula se nekada naziva *pravilo pertle*. Zaista, ako napišemo koordinate tačaka u sledećem obliku

$$\begin{array}{cc} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ a_x & a_y \end{array}$$

formula se gradi tako što se sa jednim znakom uzimaju proizvodi odozgo-naniže, sleva-udesno (to su  $a_x b_y$ ,  $b_x c_y$  i  $c_x a_y$ ), dok se sa suprotnim znakom uzimaju proizvodi odozdo-naviše, sleva udesno (to su  $a_x c_y$ ,  $c_x b_y$  i  $b_x a_y$ ), što, kada se nacрта, zaista podseća na cik-cak vezivanje pertli.

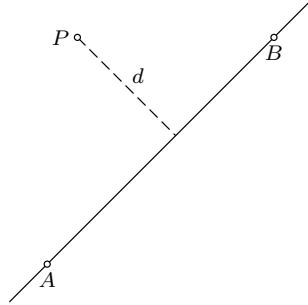
```

double površinaTrougla(Tacka A, Tacka B, Tacka C) {
    // pertle (cik-cak)
    return abs(A.x*B.y + B.x*C.y + C.x*A.y
               - A.x*C.y - C.x*B.y - B.x*A.y) / 2.0;
}

```

### Rastojanje tačke od prave

**Problem:** Odrediti rastojanje  $d$  tačke  $P$  od prave određene tačkama  $A$  i  $B$  (slika 4).



Slika 4: Rastojanje tačke od prave.

Površina trougla sa temenima  $P$ ,  $A$  i  $B$  može se izračunati na dva načina: ona je s jedne strane jednaka  $\frac{|\vec{AB}| \cdot d}{2}$  i istovremeno je jednaka  $\frac{|\vec{PA} \times \vec{PB}|}{2}$ . Dakle, rastojanje  $d$  možemo izračunati na osnovu jednačine:

$$d = \frac{|\vec{PA} \times \vec{PB}|}{|\vec{AB}|}$$

Alternativno, mogli smo odrediti i implicitnu jednačinu prave kroz tačke  $A$  i  $B$  ( $ax + by + c = 0$ ) a zatim izračunati rastojanje tačke  $P(x_0, y_0)$  od date prave po formuli:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

### Kolinearnost tačaka

**Problem:** Date su tri tačke u ravni svojim koordinatama  $A(a_x, a_y)$ ,  $B(b_x, b_y)$  i  $C(c_x, c_y)$ . Utvrditi da li su one kolinearne.

Tri tačke su kolinearne ukoliko je nagib prave određene tačkama  $A$  i  $B$  jednak nagibu prave određene tačkama  $A$  i  $C$ . Nagib prave određene tačkama  $A$  i  $B$  jednak je količniku razlike njihovih  $y$  i  $x$  koordinata:  $k_{AB} = \frac{b_y - a_y}{b_x - a_x}$ . Slično,  $k_{AC} = \frac{c_y - a_y}{c_x - a_x}$ . Dakle, tačke će biti kolinearne ukoliko važi:

$$\frac{b_y - a_y}{b_x - a_x} = \frac{c_y - a_y}{c_x - a_x}$$

odnosno:

$$(b_y - a_y) \cdot (c_x - a_x) = (c_y - a_y) \cdot (b_x - a_x)$$

```

// tacku u ravni predstavljamo njenom x i y koordinatom
struct Tacka{
    int x;
    int y;
};

// ispituje se kolinearnost tacaka
// razmatranjem nagiba po dve tacke
void kolinearne(Tacka A, Tacka B, Tacka C){
    if ((B.y - A.y) * (C.x - A.x) == (C.y - A.y) * (B.x - A.x))
        cout << "Kolinearne su" << endl;
    else
        cout << "Nisu kolinearne" << endl;
}

int main(){
    Tacka A, B, C;
    cout << "Unesi koordinate tacaka A, B i C" << endl;
    cin >> A.x >> A.y >> B.x >> B.y >> C.x >> C.y;
    kolinearne(A, B, C);
    return 0;
}

```

Primitimo da smo prilikom ispitivanja kolinearnosti tačaka vrednosti odgovarajućih izraza poredili na jednakost operatorom `==`. Ovo je korektno u slučaju kada su koordinate tačaka (a samim tim i vrednosti koje poredimo) celobrojne. Ukoliko bi koordinate tačaka bile realni brojevi, onda bi poređenje na jednakost trebalo zameniti proverom da li je apsolutna vrednost razlike vrednosti datih izraza manja od vrednosti  $\epsilon$  gde je  $\epsilon$  neka mala pozitivna vrednost.

Drugi način da ispitamo kolinearnost datih tačaka bi se zasnivao na narednom tvrđenju: tri tačke pripadaju istoj pravoj ako je površina trougla određenog ovim trima tačkama jednaka 0. Na osnovu prethodnih razmatranja, znamo da se površina  $P$  trougla  $ABC$  može izračunati po formuli:

$$P = \frac{|b_x a_y - a_x b_y + c_x b_y - b_x c_y + a_x c_y - c_x a_y|}{2}$$

Ukoliko su koordinate tačaka celobrojne, vrednost  $2P$  je takođe celobrojna i jednaka je 0 ako su tačke kolinearne, te ćemo u narednoj implementaciji pretpostaviti da temena trougla imaju celobrojne koordinate i računaćemo dvostruku označenu površinu trougla (bez deljenja sa 2 i računanja apsolutne vrednosti).

```

// funkcija za izracunavanje dvostruke oznacene površine trougla
// ako su zadata njegova temena
int dvostrukaOznacenaPovrsina(Tacka A, Tacka B, Tacka C){
    return B.x*A.y - A.x*B.y + C.x*B.y
}

```

```

        - B.x*C.y + A.x*C.y - C.x*A.y;
    }

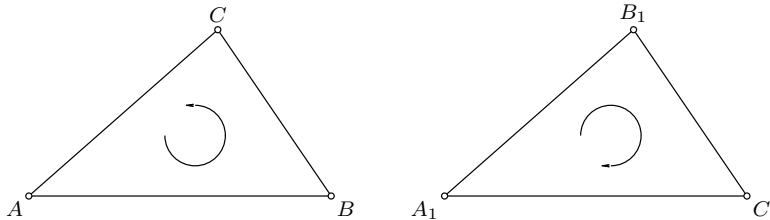
    // funkcija koja utvrdjuje da li su tacke kolinearne ili ne
    void kolinearne(Tacka A, Tacka B, Tacka C){
        int P = dvostrukaOznacenaPovrsina(A, B, C);
        if (P == 0)
            cout << "Kolinearne su" << endl;
        else
            cout << "Nisu kolinearne" << endl;
    }

```

## Orijentacija

Uređena trojka tačaka u ravni može biti kolinearna ili imati:

- orijentaciju u smeru suprotnom od kretanja kazaljke na časovniku – pozitivna orijentacija (slika 5a)
- orijentaciju u smeru kretanja kazaljke na časovniku – negativna orijentacija (slika 5b)



Slika 5: a) Trougao  $ABC$  ima matematički pozitivnu orijentaciju. b) Trougao  $A_1B_1C_1$  ima matematički negativnu orijentaciju.

Važe naredna svojstva:

- ako uređena trojka  $A, B, C$  ima pozitivnu (negativnu) orijentaciju, onda i uređena trojka  $B, C, A$  ima pozitivnu (negativnu) orijentaciju
- ako uređena trojka  $A, B, C$  ima pozitivnu (negativnu) orijentaciju, onda uređena trojka  $B, A, C$  ima negativnu (pozitivnu) orijentaciju

**Problem:** Za dati trougao  $ABC$  odrediti njegovu orijentaciju.

Važi naredno tvrđenje: trougao  $ABC$  ima pozitivnu orijentaciju ako vektorski proizvod vektora  $\overrightarrow{AB}$  i  $\overrightarrow{AC}$  ima pozitivnu vrednost.

S obzirom na to da temena  $A$ ,  $B$  i  $C$  pripadaju ravni  $Oxy$ , i vektori  $\vec{AB}$  i  $\vec{AC}$  pripadaju ovoj ravni, odnosno imaju koordinate  $\vec{AB}(x_1, y_1, 0)$  i  $\vec{AC}(x_2, y_2, 0)$ . Njihov vektorski proizvod biće vektor upravan na ravan  $Oxy$ , odnosno vektor sa koordinatama  $(0, 0, z)$ . Stoga će njegov intenzitet biti jednak apsolutnoj vrednosti koordinate  $z$  koja je jednaka  $x_1y_2 - y_1x_2$ . Dakle, ako su koordinate temena  $A(a_x, a_y, 0)$ ,  $B(b_x, b_y, 0)$  i  $C(c_x, c_y, 0)$  orijentaciju trougla  $ABC$  određujemo na osnovu znaka  $z$  koordinate vektorskog proizvoda  $\vec{AB} \times \vec{AC}$ , odnosno na osnovu vrednosti izraza:

$$d = (b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y)$$

Ukoliko je:

- $d > 0$  – trougao ima pozitivnu orijentaciju
- $d < 0$  – trougao ima negativnu orijentaciju
- $d = 0$  – tačke su kolinearne

```

struct Tacka{
    int x;
    int y;
};

enum orij {KOLINEARNE, NEGATIVNA_ORJ, POZITIVNA_ORJ};

// funkcija koja utvrđuje orijentaciju tacaka P1, P2 i P3
orij orijentacija(Tacka P1, Tacka P2, Tacka P3){
    int d = (P2.x - P1.x) * (P3.y - P1.y) - (P3.x - P1.x) * (P2.y - P1.y);
    if (d == 0) return KOLINEARNE;
    else if (d > 0)
        return POZITIVNA_ORJ;
    else return NEGATIVNA_ORJ;
}

int main(){
    Tacka A, B, C;
    cout << "Unesi koordinate tacaka A, B i C" << endl;
    cin >> A.x >> A.y >> B.x >> B.y >> C.x >> C.y;

    orij o = orijentacija(A, B, C);
    if (o == KOLINEARNE)
        cout << "Tacke A, B i C su kolinearne" << endl;
    else if (o == NEGATIVNA_ORJ)
        cout << "Trougao ABC ima negativnu orijentaciju" << endl;
    else
        cout << "Trougao ABC ima pozitivnu orijentaciju" << endl;
}

```

```

    return 0;
}

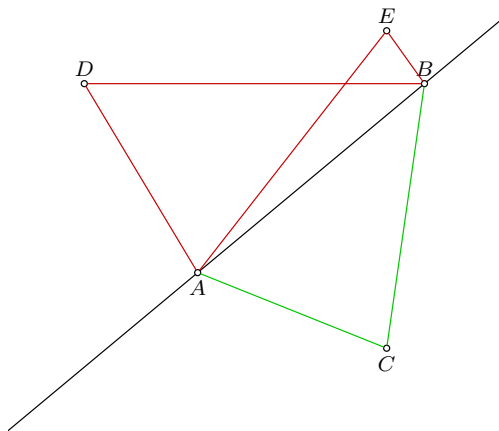
```

I ovdje se prilikom ispitivanja orijentacije tačaka podrazumeva da su koordinate tačaka celobrojne, te se proverava da li su vrednosti dva izraza jednake svodi na poređenje na jednakost primenom operatora `==`. Ukoliko tačke imaju koordinate koje su brojevi u pokretnom zarezu, i odgovarajući izrazi bi imali realne vrednosti, te bi proveru jednakosti trebalo drugačije realizovati.

### Primene pojma orijentacije

**Problem:** Za date tačke  $C$  i  $D$  utvrditi da li su sa iste strane prave  $p$  određene tačkama  $A$  i  $B$ .

Tačke  $C$  i  $D$  biće sa iste strane prave  $p$  ako i samo ako su trouglovi  $ABC$  i  $ABD$ ,  $A, B \in p$  iste orijentacije (slika 6). Dakle, dovoljno je da ispitamo da li su odgovarajući vektorski proizvodi istog znaka.



Slika 6: Tačke  $D$  i  $E$  se nalaze sa iste strane prave određene tačkama  $A$  i  $B$ , dok su tačke  $C$  i  $D$  sa različitih strana.

```

// funkcija za izracunavanje dvostruke oznacene površine trougla
// ako su zadata njegova temena
int dvostrukaOznacenaPovrsina(Tacka A, Tacka B, Tacka C){
    return A.x*B.y + B.x*C.y + C.x*A.y
           - A.x*C.y - C.x*B.y - B.x*A.y;
}

int main() {
    Tacka A, B, C, D;
}

```

```

cout << "Unesi koordinate tacaka A i B" << endl;
cin >> A.x >> A.y >> B.x >> B.y;
cout << "Unesi koordinate tacaka C i D" << endl;
cin >> C.x >> C.y >> D.x >> D.y;

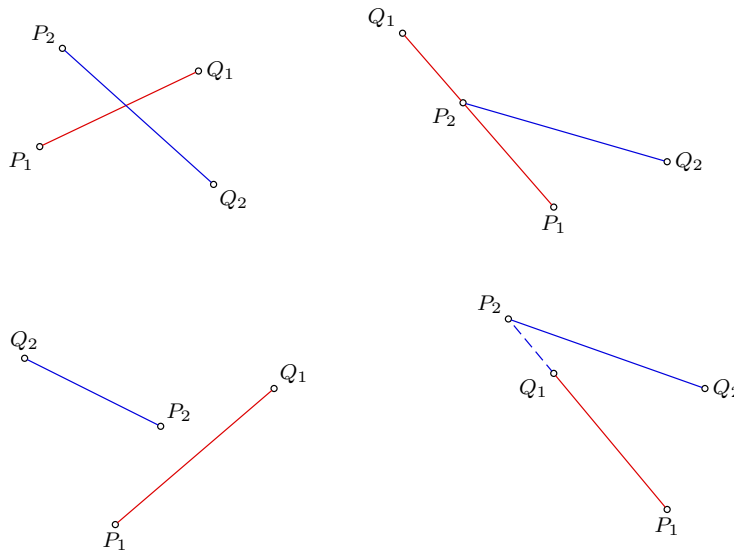
int P1 = dvostrukaOznacenaPovrsina(A,B,C);
int P2 = dvostrukaOznacenaPovrsina(A,B,D);
if ((P1 > 0 && P2 > 0) || (P1 < 0 && P2 < 0))
    cout << "Tacke se nalaze sa iste strane" << endl;
else
    cout << "Tacke se nalaze sa razlicitih strana" << endl;
return 0;
}

```

**Problem:** Date su dve duži  $P_1Q_1$  i  $P_2Q_2$  u ravni. Utvrditi da li se ove dve duži seku.

Važi sledeće tvrđenje: duži  $P_1Q_1$  i  $P_2Q_2$  se seku ako i samo ako važi jedan od sledećih uslova:

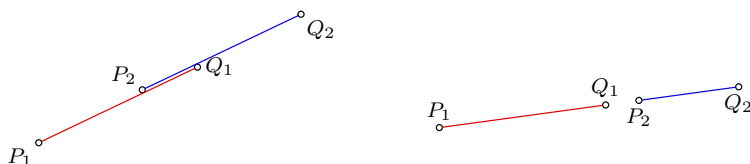
1. nikoje tri tačke nisu kolinearne i važi da su tačke  $P_2$  i  $Q_2$  sa različitih strana prave  $P_1Q_1$  i tačke  $P_1$  i  $Q_1$  su sa različitih strana prave  $P_2Q_2$  (slika 7),



Slika 7: Različiti položaji duži  $P_1Q_1$  i  $P_2Q_2$  u ravni u slučaju kada nisu sve četiri krajnje tačke duži kolinearne.



2. sve četiri navedene trojke tačaka su kolinearne i seku se i projekcije duži na  $x$  osu i projekcije duži na  $y$  osu (slika 8).



Slika 8: Različiti položaji duži  $P_1Q_1$  i  $P_2Q_2$  u ravni u slučaju kada su sve četiri krajnje tačke duži kolinearne.

Pri uslov ćemo proveriti tako što ćemo ispitati da li trojke tačaka  $P_1, Q_1, P_2$  i  $P_1, Q_1, Q_2$  imaju različitu orijentaciju i, takođe, trojke tačaka  $P_2, Q_2, P_1$  i  $P_2, Q_2, Q_1$  imaju različitu orijentaciju. Razmatraćemo, pritom, sve tri moguće različite vrednosti orijentacije – kolinearne, pozitivna i negativna orijentacija i pod taj slučaj podvesti i situaciju kada su neke tri tačke kolinearne. Drugi uslov ćemo ispitati tako što ćemo proveriti da li ako je trojka tačaka  $P_1, Q_1, P_2$  kolinearne važi da tačka  $Q_1 \in P_2Q_2$  i slično za druge trojke tačaka.

```
// za date tri kolinearne tacke P, Q i R proveravamo da li
// tacka Q pripada duzi PR
bool pripadaDuzi(Tacka P, Tacka Q, Tacka R){
    if (Q.x <= max(P.x, R.x) && Q.x >= min(P.x, R.x) &&
        Q.y <= max(P.y, R.y) && Q.y >= min(P.y, R.y))
        return true;
    return false;
}

// funkcija koja utvrdjuje orijentaciju tacaka P1, P2 i P3
orij orijentacija(Tacka P1, Tacka P2, Tacka P3){
    int d = (P2.x - P1.x) * (P3.y - P1.y) - (P3.x - P1.x) * (P2.y - P1.y);
    if (d == 0) return KOLINEARNE;
    else if (d > 0)
        return POZITIVNA_ORJ;
    else return NEGATIVNA_ORJ;
}

// proveravamo da li se duzi P1Q1 i P2Q2 seku
bool sekuSe(Tacka P1, Tacka Q1, Tacka P2, Tacka Q2){
    // racunamo vrednosti cetiri orijentacije
    // koje su nam potrebne
    orij o1 = orijentacija(P1, Q1, P2);
```

```

orij o2 = orijentacija(P1, Q1, Q2);
orij o3 = orijentacija(P2, Q2, P1);
orij o4 = orijentacija(P2, Q2, Q1);

// prvi slucaj
if (o1 != o2 && o3 != o4)
    return true;

// drugi slucaj, ako je bar jedna od trojki tacaka kolinearna,
// onda je potreban i dovoljan uslov da je
// bar jedno od temena jedne duzi izmedju temena druge duzi
// P1, Q1 i P2 su kolinearne i P2 pripada duzi P1Q1
if (o1 == KOLINEARNE && pripadaDuzi(P1, P2, Q1))
    return true;

// P1, Q1 i Q2 su kolinearne i Q2 pripada duzi P1Q1
if (o2 == KOLINEARNE && pripadaDuzi(P1, Q2, Q1))
    return true;

// P2, Q2 i P1 su kolinearne i P1 pripada duzi P2Q2
if (o3 == KOLINEARNE && pripadaDuzi(P2, P1, Q2))
    return true;

// P2, Q2 i P1 su kolinearne i Q1 pripada duzi P2Q2
if (o4 == KOLINEARNE && pripadaDuzi(P2, Q1, Q2))
    return true;

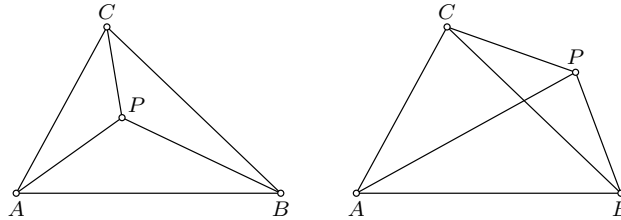
// inace se duzi ne seku
return false;
}

int main(){
    Tacka P1, Q1, P2, Q2;
    cout << "Unesi koordinate tacaka P1 i Q1" << endl;
    cin >> P1.x >> P1.y >> Q1.x >> Q1.y;
    cout << "Unesi koordinate tacaka P2 i Q2" << endl;
    cin >> P2.x >> P2.y >> Q2.x >> Q2.y;
    if (sekuSe(P1, Q1, P2, Q2))
        cout << "Duzi P1Q1 i P2Q2 se seku" << endl;
    else
        cout << "Duzi P1Q1 i P2Q2 se ne seku" << endl;
    return 0;
}

```

## Ispitivanje da li tačka pripada trouglu

**Problem:** Dat je trougao  $ABC$  i tačka  $P$ . Ispitati da li tačka  $P$  pripada trouglu  $ABC$  ili ne (slika 9).



Slika 9: Situacija kada tačka  $P$  pripada i kada ne pripada trouglu  $ABC$ .

Važi sledeće tvrđenje: ako je tačka  $P$  u unutrašnjosti trougla  $ABC$ , onda je zbir površina trouglova  $PAB$ ,  $PBC$  i  $PCA$  jednak površini trougla  $ABC$ .

```
#define EPS 1.0E-6
```

```
// funkcija kojom proveravamo da li se tacka P nalazi u trouglu ABC
```

```
bool tackaUTrouglu(Tacka A, Tacka B, Tacka C, Tacka P) {
```

```
    // racunamo površinu trougla ABC
```

```
    double P = površina(A, B, C);
```

```
    // racunamo površine trouglova PBC, PAC i PAB
```

```
    double P1 = površina(P, B, C);
```

```
    double P2 = površina(A, P, C);
```

```
    double P3 = površina(A, B, P);
```

```
    // proveravamo da li one u zbiru daju površinu trougla ABC
```

```
    // poredimo dve realne vrednosti na jednakost
```

```
    return fabs(P - (P1 + P2 + P3)) < EPS;
```

```
}
```

```
int main(){
```

```
    Tacka A, B, C, P;
```

```
    cout << "Unesi koordinate temena A, B i C" << endl;
```

```
    cin >> A.x >> A.y >> B.x >> B.y >> C.x >> C.y;
```

```
    cout << "Unesi koordinate tacke P" << endl;
```

```
    cin >> P.x >> P.y;
```

```
    if (tackaUTrouglu(A,B,C,P))
```

```
        cout << "Tacka se nalazi u trouglu" << endl;
```

```
    else
```

```

    cout << "Tacka se ne nalazi u trouglu" << endl;
    return 0;
}

```

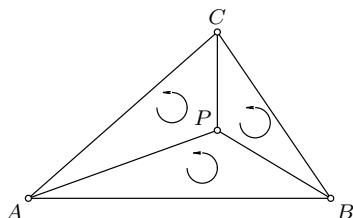
Važi, takođe, i sledeće tvrđenje: tačka  $P$  se nalazi u trouglu  $ABC$  ako i samo ako su trouglovi  $ABP$ ,  $BCP$  i  $CAP$  iste orijentacije (slika 10).

```

// funkcija kojom proveravamo da li se tacka P nalazi u trouglu ABC
bool tackaUTrouglu(Tacka A, Tacka B, Tacka C, Tacka P){
    // racunamo orijentacije sva tri trougla
    orij o1 = orijentacija(A,B,P);
    orij o2 = orijentacija(B,C,P);
    orij o3 = orijentacija(C,A,P);

    if (o1 == o2 && o2 == o3)
        return true;
    else
        return false;
}

```

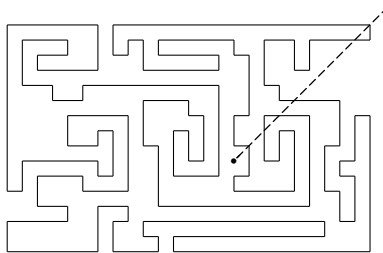


Slika 10: Orijetacije trouglova  $ABP$ ,  $BCP$  i  $CAP$  su iste kada tačka  $P$  pripada trouglu  $ABC$ .

## Ispitivanje da li tačka pripada prostom mnogouglu

**Problem** Zadat je prost mnogougao  $P$  i tačka  $Q$ . Ustanoviti da li je tačka  $Q$  u ili van mnogougla  $P$ .

Problem izgleda jednostavno na prvi pogled, ali ako se razmatraju složeni nekonveksni mnogouglovi, kao onaj na slici 11, problem sigurno nije jednostavan. Prvi intuitivni pristup je pokušati nekako “izaći napolje”, polazeći od zadate tačke  $Q$ . Posmatrajmo proizvoljnu polupravu sa temenom  $Q$ . Vidi se da je dovoljno prebrojati preseke sa stranicama mnogougla, sve do dostizanja spoljašnje oblasti. U primeru na slici 11, idući na severoistok od date tačke (prateći isprekidanu liniju), nailazimo na šest preseka sa mnogougлом do dostizanja spoljašnje oblasti.

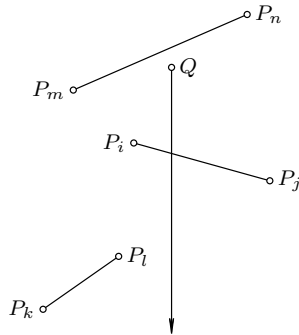


Slika 11: Utvrđivanje pripadnosti tačke unutrašnjosti prostog mnogougla.

Istina, već posle četiri preseka stiže se van mnogougla, ali to računar “ne vidi”; zato se broji ukupan broj preseka poluprave sa stranicama mnogougla. Pošto nas poslednji presek pre izlaska izvodi iz mnogougla, a pretposlednji nas vraća u mnogougao, i tako dalje, tačka  $Q$  je van mnogougla. Dakle, možemo zaključiti da je tačka  $Q$  u mnogouglu ako i samo ako je broj preseka poluprave iz  $Q$  sa stranicama mnogougla neparan.

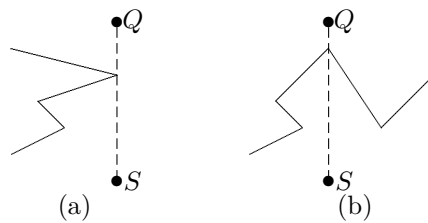
Razvijajući ovaj algoritam, implicitno smo pretpostavljali da radimo sa slikom. Problem je nešto drugačiji kad je mnogougao dat nizom koordinata temena, što je uobičajeni scenario. Na primer, kad posao radimo ručno i vidimo mnogougao, lako je naći dobar put (onaj sa malo preseka) do neke tačke van mnogougla. U slučaju mnogougla datog nizom koordinata, to nije lako. Najveći deo vremena troši se na ispitivanje preseka poluprave i stranica mnogougla. Taj deo posla može se bitno uprostiti ako je poluprava paralelna jednoj od osa — na primer  $y$ -osi i recimo usmerena nadole. Naime, presek ove poluprave kroz tačku  $Q$  sa stranicom  $P_iP_{i+1}$  postoji ako se  $x$  koordinata tačke  $Q$  nalazi između vrednosti  $x$  koordinata temena  $P_i$  i  $P_{i+1}$  i  $y$  koordinata presečne tačke poluprave kroz  $Q$  i prave  $P_iP_{i+1}$  je manja od  $y$  koordinate tačke  $Q$ .  $y$  koordinatu presečne tačke dobićemo kao presek prave  $x = x_Q$  i prave  $y - y_{P_i} = \frac{y_{P_j} - y_{P_i}}{x_{P_j} - x_{P_i}}(x - x_{P_i})$ . Primitimo da broj preseka stranica mnogougla sa ovom specijalnom polupravom može biti mnogo veći nego sa optimalnom polupravom (koju nije lako odrediti — čitaocu se ostavlja da razmotri ovaj problem), ali je nalaženje preseka mnogo jednostavnije (za konstantni faktor).

Kao što je već pomenuto, obično postoje neki specijalni slučajevi koje treba posebno razmotriti. Neka je  $S$  tačka van mnogougla i pretpostavimo da umesto poluprave iz  $Q$  razmatramo duž  $QS$ . Cilj je utvrditi da li tačka  $Q$  pripada unutrašnjosti mnogougla  $P$  na osnovu broja preseka duži  $QS$  sa stranicama mnogougla  $P$ . Naime, duž  $QS$  se može delom preklapati sa nekim stranicama mnogougla  $P$ . Ovo preklapanje očigledno ne treba brojati u preseke. Drugi specijalni slučaj je kada duž  $QS$  sadrži neko teme  $P_i$  mnogougla. Na slici 13(a) prikazan je slučaj kad presek duži  $QS$  sa temenom ne treba brojati, a na slici 13(b) slučaj kad taj presek treba brojati. Postoji više načina za utvrđivanje da li neki ovakav presek treba brojati ili ne. Na primer:



Slika 12: Različiti položaji stranica poligona i poluprave sa temenom u tački  $Q$  usmerene nadole: poluprava seče stranicu  $P_i P_j$ , a ne seče stranice  $P_k P_l$  i stranice  $P_m P_n$ .

- ako su dva temena mnogougla susedna temenu  $P_i$  sa iste strane prave kojoj pripada duž  $QS$ , presek se ne računa. U protivnom, ako su temena susedna temenu  $P_i$  sa različitih strana ove prave, presek se broji.
- s obzirom na to da razmatramo pravu koja je paralelna sa  $y$  osom, za svaku stranicu mnogougla trebalo bi računati preseke sa levim temenom, a ne sa desnim (temena neke stranice mnogougla klasifikujemo kao levo i desno u odnosu na vrednosti  $x$  koordinate temena). Time nećemo računati presek nalik onom na slici 13(a) jer je za obe stranice mnogougla susedne temenu  $P_i$  presek kroz desno teme (i slično dva puta bismo računali presek sa temenom koje je levi ekstremum i opet se ne bi promenila parnost brojača). Za presek prikazan na slici 13(b) ubrajamo jedan presek, jer je za jednu od stranica koje se susstiču u tom temenu to levo teme, a za drugu desno.



Slika 13: Specijalni slučajevi kad poluprava sadrži neko teme mnogougla.

U nastavku ćemo razmotriti implementaciju koja ne uključuje obradu specijalnih slučajeva.

```
// utvrđuje da li se tacka Q nalazi u datom mnogouglu P
// racunanjem parnosti preseka sa vertikalnom polupravom
// usmerenom nadole sa temenom u tacki Q
bool tackaUMnogouglu(vector<Tacka> P, Tacka Q){
```

```

int n = P.size();
// mnogougao mora da ima bar 3 temena
if (n < 3)
    return false;

// tacka je inicijalno van mnogougla
bool uMnogouglu = false;

// prolazimo skupom svih stranica mnogougla
for(int i = 0; i < n; i++){
    // naredno teme u poretku temena mnogougla
    int j = (i + 1) % n;
    // ukoliko se x koordinata tacke Q nalazi izmedju vrednosti
    // x koordinata krajnjih tacaka stranice P[i]P[j] i
    // y koordinata presečne tacke prave x = Q.x i prave P[i]P[j]
    // je manja od y koordinate tacke Q
    if ( Q.x > min(P[i].x,P[j].x) && Q.x < max(P[i].x,P[j].x)
        && P[i].y + (P[j].y - P[i].y)/(P[j].x - P[i].x)
            * (A.x - P[i].x) < Q.y)
        // registrujemo jos jedan presek
        uMnogouglu = !uMnogouglu;
    }
return uMnogouglu;
}

int main(){
vector<Tacka> P = {{0, 0}, {10, 1}, {12, 12}, {2, 10}};
int n = P.size();
Tacka Q = {5, 8};
if (tackaUMnogouglu(P, Q))
    cout << "Tacka se nalazi u mnogouglu" << endl;
else
    cout << "Tacka se ne nalazi u mnogouglu" << endl;
return 0;
}

```

Neka je  $n$  broj stranica mnogougla. Kroz osnovnu petlju algoritma prolazi se  $n$  puta. U svakom prolasku kroz petlju računa se presek dve prave i izvršavaju se još neke operacije za konstantno vreme. Dakle, ukupno vreme izvršavanja algoritma iznosi  $O(n)$ .

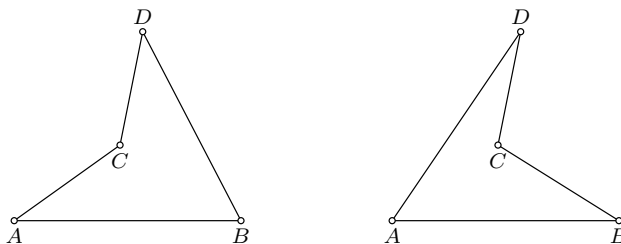
## Konstrukcija prostog mnogougla

Skup tačaka u ravni definiše veći broj različitih mnogouglova, zavisno od izabranog redosleda tačaka. Razmotrićemo sada pronalaženje prostog mnogougla sa

zadatim skupom temena.

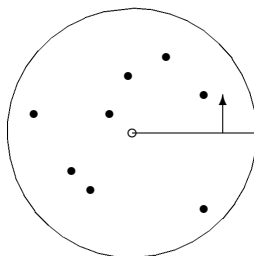
**Problem:** Dato je  $n$  tačaka u ravni, takvih da nisu sve kolinearne. Povezati ih prostim mnogouglo.

Postoji više metoda za konstrukciju traženog prostog mnogougla; uostalom, jasno je da u opštem slučaju problem nema jednoznačno rešenje (slika 14).



Slika 14: Tačke  $A$ ,  $B$ ,  $C$  i  $D$  i dva različita prosta mnogougla nad njima.

Prikažaćemo najpre geometrijski pristup ovom problemu. Neka je  $C$  neki krug, čija unutrašnjost sadrži sve tačke. Za nalaženje takvog kruga dovoljno je  $O(n)$  operacija — izračunavanja najvećeg među rastojanjima proizvoljne tačke ravni (centra kruga) do svih  $n$  tačaka. Površina  $C$  može se “prebrisati” (pregledati) rotirajućom polupravom kojoj je početak centar kruga  $C$  (slika 15).

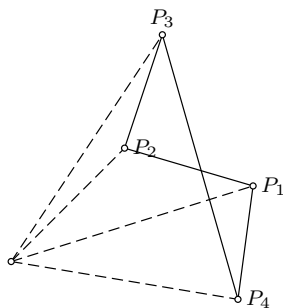


Slika 15: Prolazak tačaka u krugu rotirajućom polupravom.

Pretpostavimo za trenutak da rotirajuća poluprava u svakom trenutku sadrži najviše jednu tačku. Očekujemo da ćemo spajanjem tačaka onim redom kojim poluprava nailazi na njih dobiti prost mnogougao. Pokušajmo da to dokažemo. Označimo tačke, uređene u skladu sa redosledom nailaska poluprave na njih, sa  $P_1, P_2, \dots, P_n$  (prva tačka bira se proizvoljno). Za svako  $i$ ,  $1 \leq i \leq n$ , stranica  $P_i P_{i-1}$  (odnosno  $P_1 P_n$  za  $i = 1$ ) sadržana je u novom (disjunktном) isečku kruga, pa se ne seče ni sa jednom drugom stranom. Ako bi ovo tvrđenje bilo tačno, dobijeni mnogougao bi morao da bude prost. Međutim, ugao između polupravih kroz neke dve uzastopne tačke  $P_i$  i  $P_{i+1}$  može da bude veći od  $\pi$ . Tada isečak koji sadrži duž  $P_i P_{i+1}$  sadrži više od pola kruga i nije konveksna



figura, a duž  $P_iP_{i+1}$  prolazi kroz druge isečke kruga, pa može da seče druge stranice mnogougla. Razmotrimo primer prikazan na slici 16: duž  $P_3P_4$  prolazi kroz druge isečke kruga i seče stranicu  $P_1P_2$ .



Slika 16: Loš izbor centra kruga koji dovodi do toga da konstruisani mnogougao nije prost.

Da bi se uočeni problem rešio, mogu se, na primer, fiksirati proizvoljne tri tačke iz skupa, a za centar kruga izabrati neka tačka  $z$  unutar njima određenog trougla (na primer težište, koje se lako nalazi). Ovakav izbor garantuje da ni jedan od dobijenih sektora kruga neće imati ugao veći od  $\pi$ . Zatim tačke sortiramo prema položaju u krugu sa centrom  $z$ . Preciznije, sortiraju se *uglovi* između  $x$ -ose i polupravih od tačke  $z$  ka ostalim tačkama. Ako dve ili više tačaka imaju isti ugao, one se dalje sortiraju rastuće prema rastojanju od tačke  $z$ . Na kraju, tačke povezujemo u skladu sa dobijenim uređenjem, po dve uzastopne. Osnovna komponenta vremenske složenosti ovog algoritma potiče od sortiranja tačaka, te je složenost algoritma  $O(n \log n)$ .

Ugao  $\varphi$  koji prava  $y = mx + b$  zaklapa sa  $x$  osom dobija se korišćenjem veze  $m = \tan \varphi$ , odnosno iz jednačine  $\varphi = \arctan m$ . Primetimo da nije potrebno izračunavanje rastojanja kad dve tačke imaju isti nagib — dovoljno je izračunati kvadrate rastojanja. Dakle, nema potrebe za izračunavanjem kvadratnih korenova.

```
bool kolinearne(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
    return (t1.x - t2.x) * (t1.y - t3.y) == (t1.y - t2.y) * (t1.x - t3.x);
}
```

```
double kvadratRastojanja(double x1, double y1, double x2, double y2) {
    double dx = x1 - x2, dy = y1 - y2;
    return dx * dx + dy * dy;
}
```

```
void prostMnogougao(vector<Tacka>& tacke) {
```

```

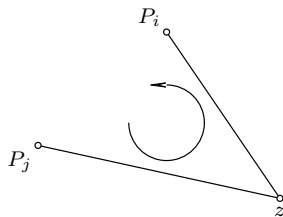
int i = 2;
// trazimo prvu tacku koja nije kolinearna sa prve dve tacke
while (kolinearne(tacke[0], tacke[1], tacke[i]))
    i++;
// racunamo koordinate centra kruga
double x0 = (tacke[0].x + tacke[1].x + tacke[i].x) / 3.0;
double y0 = (tacke[0].y + tacke[1].y + tacke[i].y) / 3.0;
// sortiramo tacke
sort(begin(tacke), end(tacke),
[x0, y0](const Tacka& t1, const Tacka& t2) {
    double x1 = t1.x - x0, y1 = t1.y - y0;
    double x2 = t2.x - x0, y2 = t2.y - y0;
    double ugao1 = atan2(y1, x1);
    double ugao2 = atan2(y2, x2);
    const double EPS = 1e-12;
    if (ugao1 < ugao2 - EPS) {
        return true;
    }
    if (ugao2 < ugao1 - EPS) {
        return false;
    }
    return kvadratRastojanja(x0, y0, x1, y1) <
        kvadratRastojanja(x0, y0, x2, y2);
});
}

int main() {
    int n;
    cin >> n;
    vector<Tacka> tacke(n);
    for (int i = 0; i < n; i++) {
        cin >> tacke[i].x >> tacke[i].y;
    }
    prostMnogougao(tacke);
    for (const Tacka& t : tacke)
        cout << t.x << " " << t.y << endl;
    return 0;
}

```

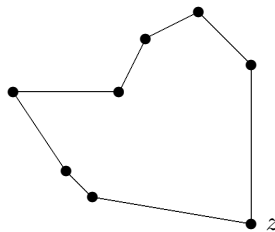
Umesto težišta trougla određenog sa neke tri nekolinearne tačke iz skupa, za centar kruga  $z$  može se uzeti i jedna tačka iz skupa – tačka sa najvećom  $x$ -koordinatom (i sa najmanjom  $y$ -koordinatom, ako ima više tačaka sa najvećom  $x$ -koordinatom). Ovakvu tačku ćemo često koristiti prilikom rešavanja geometrijskih problema i zvaćemo je *ekstremna tačka*. Ovako odabranu tačku povezujemo sa svim ostalim tačkama datog skupa i tačke sortiramo rastuće u odnosu na ugao koji poluprava od tačke  $z$  kroz tu tačku zaklapa sa  $x$ -osom. Pošto sve

tačke leže levo od tačke  $z$ , ugao između polupravih kroz dve uzastopne tačke ne može biti veći od  $\pi$ , te do degenerisanog slučaja o kome je bilo reči ne može doći. Ako dve ili više tačaka zaklapaju isti ugao sa  $x$ -osom, one se dalje sortiraju prema rastojanju od tačke  $z$  i to na sledeći način: ukoliko prvih nekoliko tačaka zaklapaju isti ugao, njih sortiramo u rastućem redosledu rastojanja od tačke  $z$ , ukoliko je poslednjih nekoliko tačaka kolinearno sa tačkom  $z$  njih sortiramo u opadajućem redosledu rastojanja od tačke  $z$ , dok je za ostale tačke koje su kolinearne sa tačkom  $z$  sve jedno da li ćemo ih sortirati rastuće ili opadajuće. Primetimo i to da je umesto računanja uglova moguće razmatrati orijentaciju tačaka. Naime za potrebe sortiranja skupa tačaka prema uglu koji zaklapaju sa horizontalnom pravom kroz tačku  $z$  iskoristićemo činjenicu da tačka  $P_i$  zaklapa manji ugao od tačke  $P_j$  ako je orijentacija trojke tačaka  $z, P_i, P_j$  pozitivna (slika 17).



Slika 17: Poređenje tačaka  $P_i$  i  $P_j$  svedeno na računanje orijentacije trojke tačaka  $z, P_i, P_j$ .

Prost mnogougao dobijen prethodno opisanim postupkom za tačke sa slike 15 prikazan je na slici 18.



Slika 18: Konstrukcija prostog mnogougla.

```
void prostMnogougao(vector<Tacka>& tacke) {
    // trazimo tacku sa maksimalnom x koordinatom,
    // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
```

```

// biramo onu sa najmanjom y koordinatom
auto max = max_element(begin(tacke), end(tacke),
    [](const Tacka& t1, const Tacka& t2) {
        return t1.x < t2.x ||
            (t1.x == t2.x && t1.y > t2.y);
    });
// ekstremnu tacku dovodimo na pocetak niza - ona predstavlja centar kruga
swap(*begin(tacke), *max);
const Tacka& t0 = tacke[0];

// sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
// zaklapaju u odnosu vertikalnu polupravu koja polazi naviše iz
// centra kruga), a kolinearne na osnovu rastojanja od centra kruga
sort(next(begin(tacke)), end(tacke),
    [t0](const Tacka& t1, const Tacka& t2) {
        orij o = orijentacija(t0, t1, t2);
        if (o == KOLINEARNE)
            return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
        return o == POZITIVNA_ORJ;
    });

// obrcemo redosled tacaka na poslednjoj pravvoj
auto it = prev(end(tacke));
while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
    it = prev(it);
reverse(it, end(tacke));
}

```