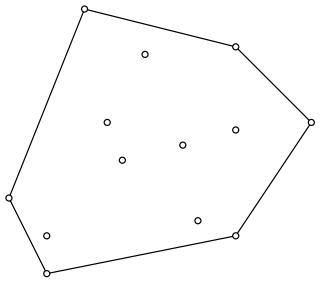
## Konveksni omotač

*Konveksni omotač* (eng. convex hull) konačnog skupa tačaka definiše se kao najmanji konveksni mnogougao koji sadrži sve tačke skupa (slika 1). Konveksni omotač se predstavlja na isti način kao običan mnogougao, redosledom svojih temena u cikličkom poretku.



Slika 1: Konveksni omotač datog skupa tačaka.

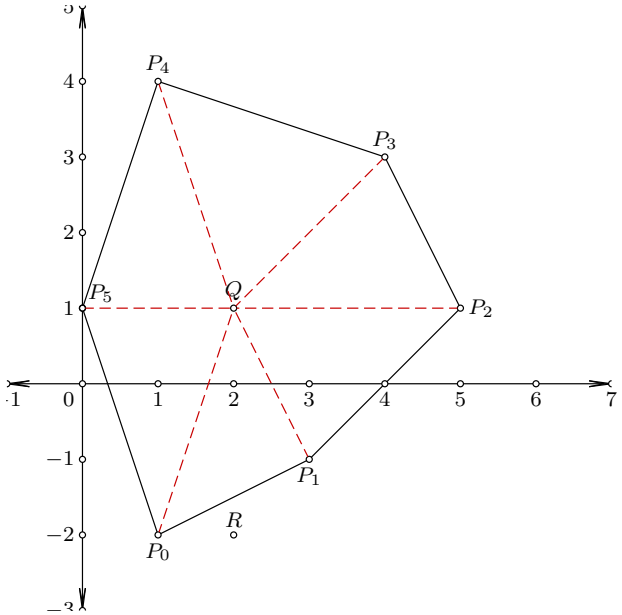
Obrada konveksnih mnogouglova jednostavnija je od obrade proizvoljnih mnogouglova. Na primer, postoji algoritam složenosti $O(\log n)$ za proveru pripadnosti tačke konveksnom $n$-touglu, dok je kao što smo već videli provera pripadnosti proizvoljnom prostom mnogouglu složenosti $O(n)$.

### Ispitivanje pripadnosti tačke konveksnom mnogouglu

**Problem:** Ispitati da li tačka $Q$ pripada konveksnom mnogouglu $P_0P_1 \dots P_{n-1}$.

Ukoliko su temena konveksnog mnogougla data u smeru suprotnom od smera kazaljke na časovniku, da bi tačka $Q$ pripadala mnogouglu dovoljno je proveriti da li se ona nalazi s leve strane svake usmerene stranice $P_iP_{i+1}$ mnogougla, odnosno da li za svako $i$ trougao $P_iP_{i+1}Q$ ima pozitivnu orijentaciju. Ova ideja predstavlja uopštenje algoritma za ispitivanje da li tačka pripada datom trouglu. U primeru prikazanom na slici 2 ovaj uslov važi za tačku $Q$, ali ne i za tačku $R$ (trougao $P_0P_1R$ ima negativnu orijentaciju). Složenost ovog algoritma iznosi $O(n)$.

```
bool uMnogouglu(vector<Tacka> tacke, Tacka Q){
  int n = tacke.size();
  for (int i = 0; i < n; i++){
    Tacka tekuca = tacke[i];
    // naredna tacka u ciklickom poretku
```




Slika 2: Ispitivanje pripadnosti tačke konveksnom mnogouglu razmatranjem orijentacije odgovarajućih trouglova.

```
    Tacka naredna = tacke[(i + 1) % n];
    // ako postoji trojka tacaka koja nema pozitivnu orijentaciju
    if (orijentacija(tekuca,naredna,Q) != POZITIVNA_ORJ)
      // tacka Q nije u mnogouglu
      return false;
  }
  // sve trojke tacaka imaju pozitivnu orijentaciju,
  // te vazi da je tacka Q u mnogouglu
  return true;
}

int main(){
  vector<Tacka> tacke = {{1,-2},{3,-1},{5,1},{4,3},{1,4},{0,1}};
  Tacka Q = {2,1};
  Tacka R = {2,-2};
  if (uMnogouglu(tacke,Q))
    cout << "Tacka Q pripada mnogouglu" << endl;
  else
    cout << "Tacka Q ne pripada mnogouglu" << endl;
  if (uMnogouglu(tacke,R))
    cout << "Tacka R pripada mnogouglu" << endl;
  else
```

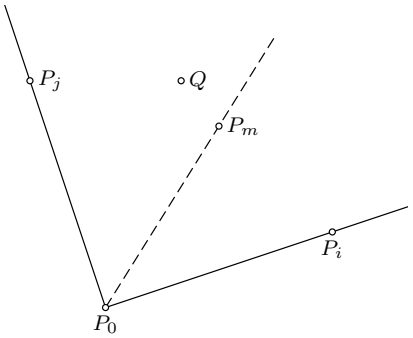
```
    cout << "Tacka R ne pripada mnogouglu" << endl;
  return 0;
}
```

Ideja na kojoj se zasniva efikasniji algoritam se oslanja na binarnu pretragu.

**Lema:** Neka se tačke $Q$ i $P_m$ nalaze unutar ugla $\angle P_i P_0 P_j$. Tada važi:

- ako su tačke $P_i$ i $Q$ sa iste strane prave $P_0P_m$, tada tačka $Q$ pripada uglu $\angle P_i P_0 P_m$
- ako su tačke $P_j$ i $Q$ sa iste strane prave $P_0P_m$, tada tačka $Q$ pripada uglu $\angle P_m P_0 P_j$

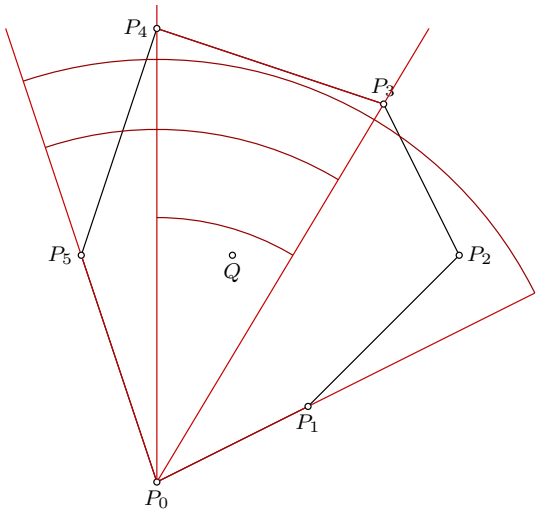
Ako tačka $Q$ ne pripada pravoj $P_0P_m$, onda je uslov da su tačke $P_j$ i $Q$ sa iste strane prave $P_0P_m$ ekvivalentan uslovu da tačke $P_i$ i $Q$ nisu sa iste strane prave $P_0P_m$. Ako tačka $Q$ baš pripada pravoj $P_0P_m$, zaključićemo da tačka $Q$ pripada uglu $\angle P_m P_0 P_j$.




Slika 3: Ilustracija leme.

Iskoristićemo prethodnu lemu da binarnom pretragom pronađemo vrednost $i$ tako da tačka $Q$ pripada uglu $P_iP_0P_{i+1}$. Održavaćemo tekući ugao $\angle P_iP_0P_j$ kome pripada tačka $Q$, koji inicijalizujemo na $\angle P_1P_0P_{n-1}$. U svakom koraku za tačku $P_m$ biramo središnju tačku – tačku sa indeksom $m = \lfloor (i+j)/2 \rfloor$ i ažuriramo ugao kome pripada tačka $Q$ prema prethodnoj lemi. Zaustavljamo se kada tačke $P_i$ i $P_j$ postanu susedna temena mnogougla. Posle najviše $O(\log n)$ koraka zaključuje se ili da tačka $Q$ ne pripada mnogouglu, ili da je unutar nekog ugla $\angle P_iP_0P_{i+1}$; u drugom slučaju tačka $Q$ je u mnogouglu akko pripada trouglu $\triangle P_iP_0P_{i+1}$. Na slici 4 prikazan je postupak polovljenja ugla kome pripada tačka $Q$; na kraju jedino preostaje da se proveri da li se tačka $Q$ nalazi unutar trougla $P_3P_0P_4$.

```
bool uMnogouglu(vector<Tacka> tacke, Tacka Q){
  int n = tacke.size();
  // proveravamo da li se tacka Q nalazi unutar ugla P_1P_0P_{n-1}
```




Slika 4: Ispitivanje pripadnosti tačke konveksnom mnogouglu metodom polovljenja ugla.

```
// tako sto proveravamo da li trojke tacaka P_0P_1Q i P_{n-1}P_0Q
// imaju pozitivnu orijentaciju
if (orijentacija(tacke[0], tacke[1], Q) != POZITIVNA_ORJ ||
    orijentacija(tacke[n-1], tacke[0], Q) != POZITIVNA_ORJ)
  return false;

// inicijalizujemo ugao
int i = 1;
int j = n-1;
// sve dok ne dodjemo do susednih temena mnogougla
while (j - i > 1){
  // racunamo indeks sredisnje tacke
  int m = (i + j)/2;
  if (saIsteStrane(tacke[0], tacke[m], Q, tacke[j])){
    // Q pripada uglu P_mP_0P_j
    i = m;
  }
  else{
    // Q pripada uglu P_iP_0P_m
    j = m;
  }
}
int p = i;
// tacka Q pripada uglu P_iP_0P_{i+1}
```

```
  // proveravamo da li pripada trouglu P_iP_{i+1}P_0
  return tackaUTrouglu(tacke[i], tacke[i+1], tacke[0], Q);
}
```

### Konstrukcija konveksnog omotača skupa tačaka

**Problem:** Konstruisati konveksni omotač zadatih $n \geq 3$ tačaka u ravni.

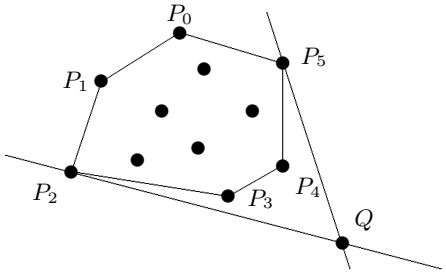
Temena konveksnog omotača su neke od tačaka iz zadatog skupa. Kazaćemo da tačka *pripada* omotaču ako je teme omotača. Koveksni omotač može se sastojati od najmanje tri, a najviše $n$ tačaka.

Konveksni omotači imaju široku primenu (na primer, u praćenju širenja epidemija, modelovanju glatkih krivih, smanjenju broja boja na slici, planiranju kretanja robota, kao gradivni element u drugim algoritmima, kao što je recimo traženje dijametra skupa tačaka), pa su zbog toga razvijeni mnogobrojni algoritmi za njihovu konstrukciju.

### Direktni induktivni pristup

Kao i obično, pokušaćemo najpre sa direktnim induktivnim pristupom. Konveksni omotač za tri tačke je lako odrediti. Pretpostavimo da umemo da konstruišemo konveksni omotač skupa od $< n$ tačaka, i pokušajmo da konstruišemo konveksni omotač skupa od $n$ tačaka. Na koji način $n$-ta tačka može da promeni konveksni omotač prvih $n - 1$ tačaka? Postoje dva moguća slučaja: ili je nova tačka u prethodnom konveksnom omotaču pa on ostaje nepromenjen, ili je ona van njega, pa se omotač "širi" da obuhvati i novu tačku (slika 5). Potrebno je dakle rešiti dva potproblema: utvrđivanje da li je nova tačka unutar omotača i proširivanje omotača novom tačkom.

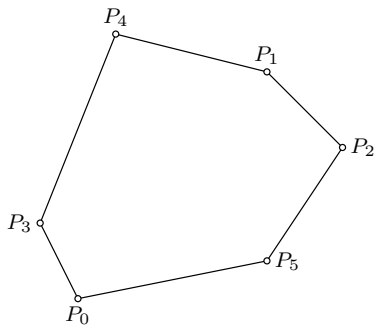
Stvar se može uprostiti pogodnim izborom $n$-te tačke. Zvuči izazovno pokušati da uvek biramo tačku unutar omotača; to, međutim, nije uvek moguće jer u nekim slučajevima sve tačke polaznog skupa pripadaju konveksnom omotaču (slika 6).




Slika 5: Proširivanje konveksnog omotača novom tačkom.

Druga mogućnost, koja se pokazala uspešnom pri rešavanju problema konstrukcije prostog mnogougla, je da se za $n$-tu tačku odabere neka ekstremna tačka, odnosno

tačka sa minimalnom ili maksimalnom $x$ koordinatom ili tačka sa minimalnom ili maksimalnom $y$ koordinatom.




Slika 6: Slučaj kada sve tačke datog skupa pripadaju konveksnom omotaču.

Na primer, izaberimo za $n$-tu tačku onu sa najvećom $x$-koordinatom (i minimalnom $y$-koordinatom ako ima više tačaka sa najvećom $x$-koordinatom). Neka je to tačka $Q$. Tačka $Q$ mora biti teme konveksnog omotača. Pitanje je kako promeniti konveksni omotač ostalih $n-1$ tačaka tako da obuhvati i tačku $Q$. Potrebno je najpre pronaći temena starog omotača koja su u unutrašnjosti novog omotača ($P_3$ i $P_4$ na slici 5) i ukloniti ih, a zatim je potrebno umetnuti novo teme $Q$ između dva postojeća ($P_2$ i $P_5$ na slici 5).

*Prava oslonca* (eng. supporting plane) konveksnog mnogougla je prava koja sadrži bar jednu tačku mnogougla i sve ostale tačke mnogougla se nalaze sa iste strane te prave. Na primer, na slici 5 prave $QP_2$ i $QP_5$ bile bi dve prave oslonca ovog mnogougla. Obično samo dva temena mnogougla povezivanjem sa $Q$ određuju prave oslonca (postoji i specijalni slučaj kad dva temena mnogougla leže na istoj pravoj sa tačkom $Q$, koji ćemo za trenutak ignorisati). Mnogougao leži između dve prave oslonca, što ukazuje na to na koji način treba izvesti modifikaciju omotača. Prave oslonca zaklapaju minimalni i maksimalni ugao sa $x$-osom među svim pravama koje sadrže tačku $Q$ i neko teme mnogougla. Da bismo odredili ta dva temena, potrebno je da razmotrimo prave iz tačke $Q$ ka svim temenima, da izračunamo uglove koje one zaklapaju sa $x$-osom, i među tim uglovima izaberemo minimalan i maksimalan. Posle pronalaženja dva ekstremna temena $P_i, P_j$ modifikovani omotač dobijamo eliminisanjem jednog od dva dobijena segmenta niza temena. Razmotrimo temena susedna temenu $P_i$. Jedno od njih je sa iste strane prave $P_iP_j$ kao i tačka $Q$; to teme pripada segmentu niza temena koje treba zameniti novim temenom $Q$. U primeru na slici 5 teme $P_4$, susedno ekstremnom temenu $P_5$, nalazi se sa iste strane prave $P_5P_2$ kao i tačka $Q$, pa se segment temena starog omotača $(P_3, P_4)$ zamenjuje novim temenom $Q$.

Razmotrimo složenost ovog algoritma. Za svaku novu tačku koja se dodaje skupu treba izračunati uglove koje grade prave određene tačkom $Q$ i svakim od temena

prethodnog omotača i $x$-osa, pronaći među njima minimalni i maksimalni ugao, izbaciti neka temena iz prethodnog omotača i dodati novo teme. Dakle, složenost dodavanja $k$-te tačke u tekući skup tačaka je $O(k)$, a ukupno razmatramo $n$ tačaka pa se složenost ovog algoritma može opisati rekurentnom jednačinom $T(n) = T(n-1) + O(n)$ čije je rešenje $O(n^2)$. Algoritam na početku zahteva i sortiranje svih tačaka skupa u neopadajućem redosledu $x$ koordinata (i opadajućem redosledu $y$ koordinata tačaka koje imaju iste vrednosti $x$ koordinata) da bi omogućio da se u svakom koraku tekućem omotaču dodaje ekstremna tačka, ali je vreme sortiranja asimptotski manje od vremena potrebnog za ostale operacije, pa ne utiče na ukupnu složenost algoritma.

```
// niz tacaka P_poc,...,P_{kraj-1} menjamo tackom P
void zameni(vector<Tacka> &omotac, int poc, int kraj, Tacka P){

  if (poc < kraj){
    // ako je poc < kraj, onda su tacke koje brisemo susedne
    omotac.erase(omotac.begin() + poc, omotac.begin() + kraj);
  } else if (poc > kraj){
     // ako je poc > kraj, onda elemente brisemo iz dva puta
     omotac.erase(omotac.begin() + poc, omotac.end());
     omotac.erase(omotac.begin(), omotac.begin() + kraj);
  }
    // ako je poc = kraj ne brisemo nista

  // dodajemo novo teme u omotac na odgovarajuce mesto
  omotac.insert(omotac.begin() + poc, P);
}

// funkcija poredjenja potrebna za bibliotecku funkciju sort()
bool poredi(const Tacka &p1, const Tacka &p2){
  return (p1.x < p2.x) || (p1.x == p2.x && p1.y > p2.y);
}

void konveksniOmotac(vector<Tacka> tacke){

  int n = tacke.size();
  // za konveksni omotac moraju postojati bar 3 tacke
  if (n < 3) return;

  // sortiramo tacke po x koordinati neopadajuce,
  // ako postoje dve tacke sa istom vrednoscu x koordinate
  // prednost dajemo onoj sa vecom y koordinatom
  sort(begin(tacke), end(tacke), poredi);

  vector<Tacka> omotac;
  // prve tri tacke dodajemo u redosledu pozitivne orijentacije u omotac
```

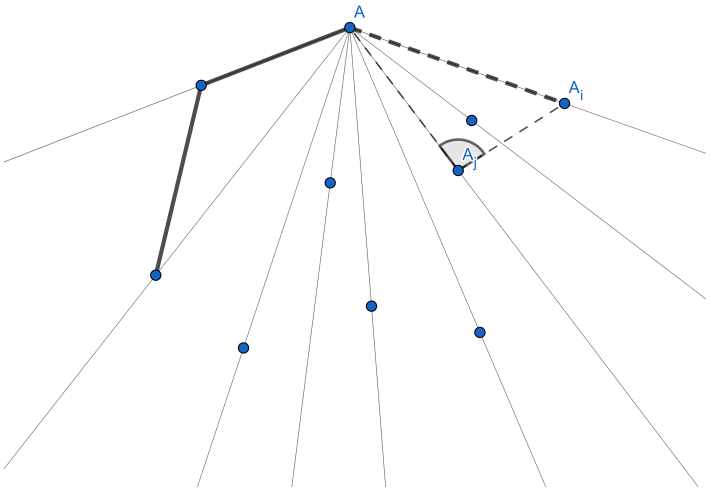
```
if (orijentacija(tacke[0], tacke[1], tacke[2]) == POZITIVNA_ORJ){
  omotac.push_back(tacke[0]);
  omotac.push_back(tacke[1]);
  omotac.push_back(tacke[2]);
}
else{
  omotac.push_back(tacke[0]);
  omotac.push_back(tacke[2]);
  omotac.push_back(tacke[1]);
}

// dodajemo jednu po jednu tacku u omotac u redosledu rastucih x koordinata
for (int k = 3; k < n; k++){

  // velicina tekuceg konveksnog omotaca
  int m = omotac.size();

  // trazimo gornju pravu oslonca
  // to je prava P_kP_i tako da za svako i1 vazi
  // da trojka tacaka {P_k,P_i,P_i1} ima pozitivnu orijentaciju
  int i = 0;
  // ako je orijentacija tacaka (P_k,P_i,P_i1) negativna
  // ili su tacke kolinearne ali je tacka P_i dalja od P_k nego P_i1
  // menjamo vrednost i vrednoscu i1;
  // tacka P_kP_i1 postaje novi kandidat za pravu oslonca
  for (int i1 = 0; i1 < m; i1++){
    if ((orijentacija(tacke[k], omotac[i], omotac[i1]) == NEGATIVNA_ORJ) ||
        (orijentacija(tacke[k], omotac[i], omotac[i1]) == KOLINEARNE
         && kvadratRastojanja(tacke[k],tacke[i]) >
            kvadratRastojanja(tacke[k],tacke[i1]))){
       i = i1;
  }

  // trazimo donju pravu oslonca
  // to je prava P_kP_j tako da za svako i1 vazi
  // da trojka tacaka {P_k,P_i1,P_j} ima pozitivnu orijentaciju
  int j = 0;
  // ako je orijentacija tacaka (P_k,P_i1,P_j) negativna
  // ili su tacke kolinearne ali je tacka P_j dalja od P_k nego P_i1
  // menjamo vrednost j vrednoscu i1;
  // tacka P_kP_i1 postaje novi kandidat za pravu oslonca
  for (int i1 = 0; i1 < m; i1++){
    if ((orijentacija(tacke[k], omotac[i1], omotac[j]) == NEGATIVNA_ORJ) ||
        (orijentacija(tacke[k], omotac[i1], omotac[j]) == KOLINEARNE
         && kvadratRastojanja(tacke[k],tacke[j]) >
            kvadratRastojanja(tacke[k],tacke[i1]))){
```

```
        j = i1;
    }

    // proveravamo sa koje strane prave PiPj se nalazi tacka P_{i-1}
    if (saIsteStrane(omotac[i], omotac[j], omotac[i-1], tacke[k])){
      // potrebno je zameniti niz tacaka P_{j+1},...,P_{i-1} sa P_k
      zameni(omotac, (j+1) % m, i, tacke[k]);
    } else if (i != (j+1) % m){
      // potrebno je zameniti niz tacaka P_{i+1},...,P_{j-1} sa P_k
      zameni(omotac, (i+1) % m, j, tacke[k]);
    } else{
      // temena P_i i P_j su susedna te samo
      // dodajemo tacku P_k na odgovarajuce mesto
      zameni(omotac, i, i, tacke[k]);
    }
  }

  // stampamo konveksni omotac
  cout << "Konveksni omotac se sastoji od tacaka" << endl;
  for (int k = 0; k < omotac.size(); k++)
    cout << "(" << omotac[k].x << ", "
         << omotac[k].y << ")" << endl;
}

int main(){
  vector<Tacka> tacke = {{0,3},{1,1},{2,2},{4,4},{0,0},
                         {1,2},{3,1},{3,3}};
  konveksniOmotac(tacke);
  return 0;
}
```

### Uvijanje poklona

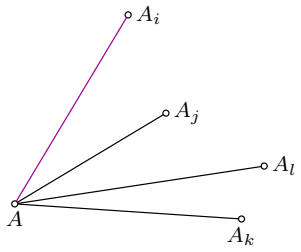
Kako se može poboljšati opisani algoritam? Kad proširujemo mnogougao teme po teme, dosta vremena trošimo na formiranje konveksnih omotača od tačaka koje mogu biti unutrašnje za konačni konveksni omotač. Može li se to izbeći? Umesto da pravimo konveksne omotače podskupova datog skupa tačaka, možemo da posmatramo kompletan skup tačaka i da direktno pravimo njegov konveksni omotač. Može se, kao i u prethodnom algoritmu, krenuti od ekstremne tačke, na primer one sa najmanjom vrednošću $x$ koordinate, a ako ih ima više one među njima koja ima najmanju $y$ koordinatu. Kao što smo već pomenuli, ekstremna tačka uvek pripada konveksnom omotaču. Ideja je da se u svakom koraku u omotač doda naredno teme koje je deo konačnog konveksnog omotača. Kako nalazimo narednu stranicu konveksnog omotača? Jedno teme te stranice biće poslednja tačka do sada određenog dela konveksnog omotača. Da bismo odredili drugo teme stranice, analiziraćemo sve tačke i tražiti onu koja daje duž koja sa

prethodnom duži gradi što veći ugao; u polaznom slučaju nema prethodne duži, pa bismo tražili tačku koja gradi najveći ugao sa negativnim delom $y$ ose (slika 7).




Slika 7: Algoritam uvijanje poklona.

Primetimo da nema potrebe za eksplicitnim računanjem uglova. Naime, ako je prethodno teme omotača $A$, tada tačka $A_i$ zaklapa veći ugao od tačke $A_j$ ako i samo ako je orijentacija trojke $(A, A_i, A_j)$ negativna (slika 8).




Slika 8: Ilustracija odabira naredne tačke omotača.

Potrebno je još obratiti pažnju na slučaj kolinearnih tačaka. Naime, ako postoji više tačaka koje određuju maksimalni ugao, tada se na toj polupravoj uzima ona koja je najdalja od prethodnog temena omotača $A$. Postupak se završava

kada se ustanovi da je tačka koja određuje najveći ugao sa prethodnom tačkom jednaka početnoj tački.

Ovaj algoritam se iz razumljivih razloga zove *uvijanje poklona* (eng. gift wrapping algorithm). Polazi se od jednog temena "poklona", i onda se on uvija u konveksni omotač pronalaženjem temena po temena omotača. Poznat je i pod nazivom *Džarvisov marš* (eng. Jarvis March algorithm), po svom autoru.

```
bool izmedju(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
  return (t1.x < t2.x && t2.x < t3.x) ||
         (t1.x > t2.x && t2.x > t3.x) ||
         (t1.x == t2.x && t2.x == t3.x && t1.y < t2.y && t2.y < t3.y) ||
         (t1.x == t2.x && t2.x == t3.x && t1.y > t2.y && t2.y > t3.y);
}

vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  vector<Tacka> omotac;

  // funkcija poredjenja potrebna za poziv funkcije min_element
  auto cmp = [] (const Tacka& t1, const Tacka& t2) {
    return t1.x < t2.x || (t1.x == t2.x && t1.y < t2.y);
  };

  // racunamo indeks ekstremne tacke i dodajemo je u omotac
  // ekstremna tacka je ovde tacka sa minimalnom x koordinatom
  int prva = distance(begin(tacke), min_element(begin(tacke), end(tacke), cmp));
  int tekuca = prva;
  do {
    omotac.push_back(tacke[tekuca]);
    int naredna = 0;
    // trazimo tacku naredna tako da za svako i vazi da je
    // orijentacija trojke {tacke[tekuca],tacke[naredna],tacke[i]} negativna
    // ili vazi da su kolinearne ali da
    // tacke[naredna] nije izmedju tacke[tekuca] i tacke[i]
    for (size_t i = 1; i < tacke.size(); i++) {
      orij o = orijentacija(tacke[tekuca], tacke[naredna], tacke[i]);
      if (naredna == tekuca ||
          o == POZITIVNA_ORJ ||
          (o == KOLINEARNE && izmedju(tacke[tekuca], tacke[naredna], tacke[i])))
        naredna = i;
    }
    tekuca = naredna;
    // ponavljamo postupak sve dok se ne vratimo na prvu tacku omotaca
  } while (tekuca != prva);
  return omotac;
}
```

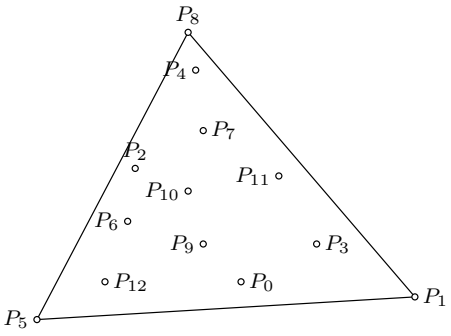
```
int main() {
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++)
    cin >> tacke[i].x >> tacke[i].y;

  vector<Tacka> omotac = konveksniOmotac(tacke);
  cout << omotac.size() << endl;
  for (const Tacka& t : omotac)
    cout << t.x << " " << t.y << endl;
  return 0;
}
```

Algoritam uvijanje poklona je direktna posledica primene sledeće induktivne hipoteze (po $k$):

**Induktivna hipoteza:** Za zadati skup od $n$ tačaka u ravni, umemo da pronađemo konveksni put dužine $k < n$ koji je deo konačnog konveksnog omotača datog skupa tačaka.

Kod ove hipoteze naglasak je na proširivanju *puta*, a ne omotača. Umesto da pronalazimo konveksne omotače podskupova, mi pronalazimo deo konačnog konveksnog omotača.

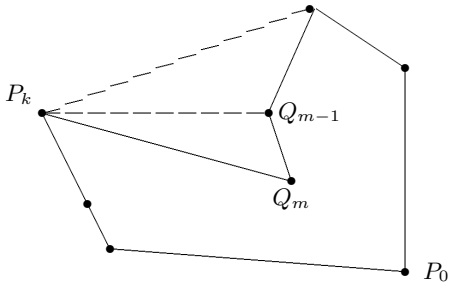
Ako je $m$ broj temena konačnog konveksnog omotača, vremenska složenost algoritma uvijanje poklona je $O(mn)$. Dakle, ovaj algoritam spada u grupu algoritama čija složenost zavisi ne samo od dimenzije ulaza, već i od dimenzije izlaza. U slučaju kada je konveksni omotač skupa tačaka trougao (slika 9) algoritam uvijanja poklona imaće linearnu vremensku složenost, a u slučaju kada sve tačke pripadaju konveksnom omotaču (slika 6) složenost algoritma biće $O(n^2)$.




Slika 9: Slučaj kada je konveksni omotač skupa tačaka trougao.

**Grejemov algoritam**

Sada ćemo razmotriti algoritam za nalaženje konveksnog omotača složenosti $O(n \log n)$. Započinje se sortiranjem tačaka prema uglovima, slično kao pri konstrukciji prostog mnogougla. Neka je $P_0$ tačka sa najvećom $x$-koordinatom (i sa najmanjom $y$-koordinatom, ako ima više tačaka sa najvećom $x$-koordinatom). Za svaku tačku $P_i$ iz datog skupa tačaka izračunavamo ugao između prave $P_0P_i$ i $x$-ose, i sortiramo tačke prema veličini ovih uglova (videti sliku 10). Tačke prolazimo redosledom kojim se pojavljuju u (prostom) mnogouglu, i prilikom obilaska pokušavamo da identifikujemo temena konveksnog omotača. Kao i kod algoritma uvijanje poklona pamtimo put sastavljen od dela prođenih tačaka. Preciznije, to je konveksni put čiji konveksni mnogougao sadrži sve do sada pregledane tačke (odgovarajući konveksni mnogougao dobija se povezivanjem prve i poslednje tačke puta). Zbog toga, u trenutku kad su sve tačke pregledane, konveksni omotač skupa tačaka je konstruisan. Osnovna razlika između ovog algoritma i algoritma uvijanja poklona je u činjenici da tekući konveksni put ne mora da bude deo konačnog konveksnog omotača. To je samo deo konveksnog omotača do sada pregledanih tačaka. Dakle tekući put može da sadrži tačke koje ne pripadaju konačnom konveksnom omotaču; te tačke biće eliminisane kasnije. Na primer, put od $P_0$ do $Q_m$ na slici 10 je konveksan, ali tačke $Q_m$ i $Q_{m-1}$ očigledno ne pripadaju konveksnom omotaču kompletnog skupa tačaka. Ovo razmatranje sugeriše algoritam zasnovan na sledećoj induktivnoj hipotezi.




Slika 10: Grejemov algoritam za nalaženje konveksnog omotača skupa tačaka.

**Induktivna hipoteza:** Ako je dato $n$ tačaka u ravni, uređenih prema algoritmu za konstrukciju prostog mnogougla, onda umemo da konstruišemo konveksni put preko nekih od prvih $k$ tačaka, takav da odgovarajući konveksni mnogougao obuhvata prvih $k$ tačaka.

Slučaj $k = 1$ je trivijalan. Označimo konveksni put dobijen (induktivno) za prvih $k$ tačaka sa $P = Q_0, Q_1, \ldots, Q_m$. Proširimo induktivnu hipotezu na $k + 1$ tačaka. Posmatrajmo ugao između pravih $Q_{m-1}Q_m$ i $Q_mP_k$ (videti sliku 10). Ako je taj ugao manji od $\pi$ (pri čemu se ugao meri iz unutrašnjosti mnogougla), onda se tačka $P_k$ može dodati postojećem putu (novi put je zbog toga takođe konveksan), čime je korak indukcije završen. U protivnom, tvrdimo da tačka $Q_m$ leži u mnogouglu dobijenom zamenom tačke $Q_m$ u putu $P$ tačkom $P_k$, i

povezivanjem tačke $P_k$ sa tačkom $P_0$. Ovo je tačno jer su tačke uređene prema odgovarajućim uglovima, pa se prava $P_0P_k$ nalazi "levo" od prvih $k$ tačaka. Zbog toga $Q_m$ jeste unutar gore definisanog mnogougla, može se izbaciti iz puta $P$, a tačka $P_k$ se može dodati tekućem putu. Da li je time završena obrada (k+1)-ve tačke? Ne sasvim. Nakon izbacivanja tačke $Q_m$ dobijeni put ne mora uvek da bude konveksan. Zaista, slika 10 jasno pokazuje da postoje slučajevi kad treba eliminisati još tačaka. Na primer, tačka $Q_{m-1}$ može da bude unutar mnogougla definisanog modifikovanim putem. Moramo da (unazad) nastavimo sa proverama poslednje dve stranice puta, sve dok ugao između njih ne postane manji od $\pi$: ovo se uvek mora desiti pre nego što se iscrpu sve tačke jer će u najgorem slučaju barem tačke $Q_0$, $Q_1$ i $P_k$ činiti konveksni put (zbog prethodnog uređenja tačaka kao u algoritmu `Prost_mnogougao`). Put je tada konveksan, a hipoteza je proširena na $k+1$ tačku.

Umesto da računamo ugao između pravih $Q_{m-1}Q_m$ i $Q_mP_k$, razmatraćemo orijentaciju trougla $Q_{m-1}Q_mP_k$: ukoliko je orijentacija ovog trougla pozitivna onda je ugao između pravih $Q_{m-1}Q_m$ i $Q_mP_k$ manji od $\pi$.

```
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);
                         });
  // dovodimo je na početak 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 pravoj
  auto it = prev(end(tacke));
  while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
    it = prev(it);
  reverse(it, end(tacke));
```

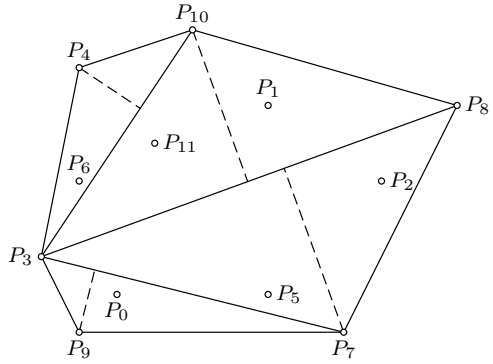
```
}

vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  vector<Tacka> omotac;
  // konstruisemo prost mnogougao nad datim skupom tacaka
  prostMnogougao(tacke);
  // u omotac dodajemo prve dve tacke
  omotac.push_back(tacke[0]);
  omotac.push_back(tacke[1]);
  for (size_t i = 2; i < tacke.size(); i++) {
    // ukoliko orijentacija tacaka nije odgovarajuca
    // iz omotaca izbacujemo poslednju tacku dodatu u omotac
    while (omotac.size() >= 2 &&
           orijentacija(omotac[omotac.size() - 2],
                        omotac[omotac.size() - 1],
                        tacke[i]) != POZITIVNA_ORJ)
      omotac.pop_back();
    // tekucu tacku dodajemo u omotac
    omotac.push_back(tacke[i]);
  }
  return omotac;
}

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

Glavni deo složenosti Grejemovog algoritma potiče od početnog sortiranja. Ostatak algoritma izvršava se za vreme $O(n)$. Svaka tačka skupa razmatra se tačno jednom u induktivnom koraku kao $P_k$. U tom trenutku tačka se uvek dodaje konveksnom putu. Ista tačka biće razmatrana i kasnije (možda čak i više nego jednom) da bi se proverila njena pripadnost konveksnom putu. Broj tačaka na koje se primenjuje ovakav povratni test može biti veliki, ali se sve one, sem dve (tekuća tačka i tačka za koju se ispostavlja da dalje pripada konveksnom putu) eliminišu, a tačka može biti eliminisana samo jednom! Prema tome, troši se najviše konstantno vreme za eliminaciju svake tačke i konstantno vreme za njeno dodavanje, te je ukupna složenost ove faze $O(n)$. Zbog početnog sortiranja

tačaka vreme izvršavanja kompletnog algoritma iznosi $O(n \log n)$.

**Brzi algoritam za traženje konveksnog omotača**

Pokušajmo da isti problem rešimo tehnikom dekompozicije, nalik algoritmu brzog sortiranja (QuickSort). Odredimo tačke iz datog skupa sa minimalnom i maksimalnom $x$ koordinatom (ako ima više takvih onda biramo onu sa minimalnom ili maksimalnom $y$ koordinatom): neka su to tačke $P_i$ i $P_j$. Obe ove tačke sigurno pripadaju konveksnom omotaču. Duž $P_iP_j$ deli skup tačaka na dva podskupa: svaki od podskupova čine tačke koje se nalaze sa iste strane te duži. Razmotrimo jedan od ova dva podskupa: u njemu tražimo tačku na maksimalnom rastojanju od duži $P_iP_j$: neka je to tačka $P_k$. I ona sigurno pripada konveksnom omotaču. Tačke iz skupa koje pripadaju trouglu $P_iP_jP_k$ ne mogu biti deo konveksnog omotača i ne moraju se dalje razmatrati. Prethodno razmatranje sada možemo primeniti na dve nove duži koje formiraju trougao: $P_iP_k$ i $P_kP_j$: tražimo najudaljeniju tačku od duži $P_iP_k$ sa one strane prave $P_iP_k$ sa koje nije $P_j$ i, slično, najudaljeniju tačku od duži $P_kP_j$ sa one strane prave $P_kP_j$ sa koje nije tačka $P_i$. Sa postupkom nastavljamo sve dok na raspolaganju ima još tačaka. Na kraju tačke koje su tokom izvršavanja algoritma birane kao najudaljenije pripadaju konveksnom omotaču. Na ovaj način došli smo do tzv. *brzog algoritma za određivanje konveksnog omotača* (eng. QuickHull algorithm).




Slika 11: Konstrukcija konveksnog omotača brzim algoritmom.

Razmotrimo primer sa slike 11: u konveksni omotač najpre dodajemo tačke $P_3$ i $P_8$ kao tačke sa minimalnom i maksimalnom $x$-koordinatom. Nakon toga određujemo najudaljenije tačke od duži $P_3P_8$ sa obe strane te duži: to su tačke $P_{10}$ i $P_7$. U nastavku algoritma tražimo najudaljeniju tačku od duži $P_3P_{10}$ sa strane sa koje nije $P_8$ – to je tačka $P_4$, dok ne postoji nijedna tačka sa one strane duži $P_{10}P_8$ sa koje nije $P_3$. Analogno, razmatranjem tačaka sa one strane duži $P_3P_7$ sa koje nije $P_8$, dobijamo da je najudaljenija tačka od duži $P_3P_7$ tačka $P_9$ i konveksnom omotaču dodajemo tačku $P_9$, dok zaključujemo da ne postoji nijedna tačka sa one strane duži $P_7P_8$ sa koje nije tačka $P_3$. Na ovaj

način dobijamo konačni konveksni omotač datog skupa tačaka, koji se sastoji od tačaka $P_3, P_8, P_{10}, P_7, P_4$ i $P_9$ u cikličkom rasporedu.

Složenost ovog algoritma može se opisati rekurentnom jednačinom oblika $T(n) = T(k) + T(n-k-1) + O(n)$, gde je sa $k$ označena veličina jednog, a sa $n-k-1$ veličina drugog problema na koji se vrši podela (nalik kviksort algoritmu). U najboljem slučaju problem se deli na dva potproblema iste dimenzije pa dobijamo rekurentnu jednačinu $T(n) = 2T(n/2) + O(n)$ čije je rešenje $T(n) = O(n \log n)$. U najgorem slučaju dobijamo rekurentnu jednačinu oblika $T(n) = T(n-1)+O(n)$ čije rešenje zadovoljava jednačinu $T(n) = O(n^2)$.

Napomenimo da prilikom traženja najudaljenije tačke $P_k$ od duži $P_iP_j$, nije neophodno računati tačno rastojanje po formuli:

$$d = \frac{|\overrightarrow{P_kP_i} \times \overrightarrow{P_kP_j}|}{|\overrightarrow{P_iP_j}|}$$

Naime, mi tražimo najudaljeniju tačku od prave kroz fiksirane dve tačke, te će imenilac ovog izraza uvek biti konstantan. Stoga ćemo meru rastojanja ovde računati po formuli $d = |\overrightarrow{P_kP_i} \times \overrightarrow{P_kP_j}|$.

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

// posto cemo konveksni omotac pamtiti u vidu skupa tacaka
// neophodno je da definisemo operator < za dve tacke
bool operator<(const Tacka& A, const Tacka&B){
  return A.x < B.x || (A.x == B.x && A.y < B.y);
}

// racuna se kvadrat rastojanja izmedju tacaka A i B
int kvadratRastojanja(Tacka A, Tacka B){
    return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y);
}

// orijentacija uredjene trojke tacaka P,Q i R
int orijentacija(Tacka P, Tacka Q, Tacka R){

  int d = (Q.y - P.y) * (R.x - Q.x) - (Q.x - P.x) * (R.y - Q.y);
  if (d == 0)
    // kolinearne su
    return KOLINEARNE;
  else if (d > 0)
    return NEGATIVNA_ORJ;
```

```
  else
    return POZITIVNA_ORJ;
}

// funkcija koja vraca vrednost koja je proporcionalna rastojanju
// od tacke A do duzi odredjene tackama P i Q
int rastojanjeOdDuzi(Tacka P, Tacka Q, Tacka A){
  return abs((A.y - P.y) * (Q.x - P.x) - (Q.y - P.y) * (A.x - P.x));
}

// funkcija poredjenja koja je potrebna za bibliotecke funkcije
// min_element i max_element
bool poredi(Tacka A, Tacka B){
  return (A.x < B.x) || (A.x == B.x && A.y > B.y);
}

// trazimo konveksni omotac tacaka sa jedne strane duzi PQ
// parametar `strana` moze imati vrednost 1 ili -1
void konveksniOmotac(vector<Tacka> tacke, Tacka P, Tacka Q,
     int strana, set<Tacka> &omotac){

  int n = tacke.size();
  int ind = -1;
  int dMax = 0;

  // trazimo tacku sa date strane duzi
  // koja je na najvecem rastojanju od te duzi
  for (int i = 0; i < n; i++){

    int dTekuce = rastojanjeOdDuzi(P,Q,tacke[i]);
    if (orijentacija(P,Q,tacke[i]) == strana && dTekuce > dMax){
      ind = i;
      dMax = dTekuce;
    }
  }

  // ako ne postoji tacka sa date strane
  // dodajemo krajnje tacke P i Q duzi i zavrsavamo
  if (ind == -1){
    omotac.insert(P);
    omotac.insert(Q);
    return;
  }

  // rekurzivno pozivamo za dva podskupa dobijena tackom `tacke[ind]`
  // trazimo najudaljeniju tacku od prave T_{ind}P na strani suprotnoj od Q
```

```
  konveksniOmotac(tacke,tacke[ind],P,-orijentacija(tacke[ind],P,Q),omotac);
  // trazimo najudaljeniju tacku od prave T_{ind}Q na strani suprotnoj od P
  konveksniOmotac(tacke,tacke[ind],Q,-orijentacija(tacke[ind],Q,P),omotac);
}

// funkcija koja racuna konveksni omotac skupa tacaka
void konveksniOmotac(vector<Tacka> tacke){
  int n = tacke.size();
  // za konveksni omotac mora postojati bar 3 tacke
  if (n < 3) return;

  // omotac pamtimo u vidu skupa
  set<Tacka> omotac;

  // trazimo tacke sa najmanjom i najvecom x koordinatom
  int min = min_element(tacke.begin(),tacke.end(),poredi)
            - tacke.begin();
  int max = max_element(tacke.begin(),tacke.end(),poredi)
            - tacke.begin();

  // rekurzivno trazimo konveksne omotace tacaka sa jedne strane
  // duzi odredjene tackama tacka[min] i tacka[max]
  konveksniOmotac(tacke,tacke[min],tacke[max],1,omotac);
  // i sa druge strane
  konveksniOmotac(tacke,tacke[min],tacke[max],-1,omotac);

  // stampamo omotac
  cout << "Konveksni omotac se sastoji od tacaka" << endl;
  for (auto it = omotac.begin(); it != omotac.end(); it++){
    Tacka prva = *it;
    cout << "(" << prva.x << ", " << prva.y << ")" << endl;
    omotac.erase(omotac.begin());
  }
}
```

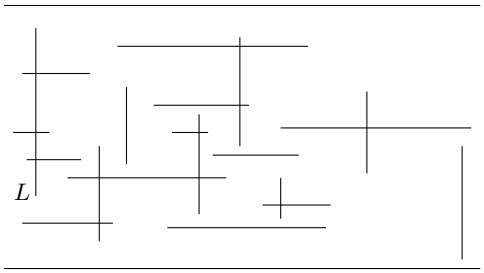
## Preseci horizontalnih i vertikalnih duži

Često se nailazi na probleme nalaženja preseka. Nekad je potrebno pronaći preseke više objekata, a ponekad samo treba otkriti da li je presek neprazan skup. U nastavku ćemo prikazati jedan problem nalaženja preseka, koji ilustruje važnu tehniku za rešavanje geometrijskih problema. Ista tehnika može se primeniti i na druge slične probleme.

**Problem:** Za zadati skup od $n$ horizontalnih i $m$ vertikalnih duži pronaći sve njihove preseke.

Ovaj problem važan je, na primer, pri projektovanju kola VLSI (integrisanih

kola sa ogromnim brojem elemenata). Kolo može da sadrži na hiljade "žičica", a projektant treba da bude siguran da ne postoje neočekivani preseci. Na problem se takođe nailazi pri eliminaciji skrivenih linija kada je potrebno zaključiti koje stranice ili delovi stranica su skriveni samim objektom ili nekim drugim objektom; taj problem je obično komplikovaniji, jer se ne radi samo o horizontalnim i vertikalnim linijama. Primer ulaza za ovaj problem prikazan je na slici 12.




Slika 12: Preseci horizontalnih i vertikalnih duži.

Nalaženje svih preseka samo vertikalnih duži je jednostavan problem, koji se ostavlja čitaocu za vežbanje (isto se odnosi i na horizontalne duži). Pretpostavimo zbog jednostavnosti da ne postoje preseci između proizvoljne dve horizontalne, odnosno proizvoljne dve vertikalne duži. Ako pokušamo da problem rešimo uklanjanjem jedne po jedne duži (bilo horizontalne, bilo vertikalne), onda će biti neophodno da se pronađu preseci uklonjene duži sa svim ostalim dužima, pa se dobija algoritam sa $O(mn)$ nalaženja preseka duži. U opštem slučaju broj preseka može da bude $O(mn)$, pa algoritam može da utroši vreme $O(mn)$ već samo za prikazivanje svih preseka. Međutim, broj preseka može da bude mnogo manji od $mn$. Voleli bismo da konstruišemo algoritam koji radi dobro kad ima malo preseka, a ne previše loše ako preseka ima mnogo. Dakle cilj nam je da problem rešimo korišćenjem algoritma osetljivog na izlaz, čija složenost zavisi i od veličine ulaza i od veličine izlaza. To se može postići kombinovanjem dveju ideja: izbora specijalnog redosleda indukcije i pojačavanja induktivne hipoteze.

Redosled primene indukcije može se odrediti *pokretnom pravom* (eng. sweeping line) koja prelazi ("skenira") ravan sleva udesno; duži se razmatraju onim redom kojim na njih nailazi pokretna prava. Pored nalaženja presečnih tačaka, treba čuvati i neke podatke o dužima koje je pokretna prava već zahvatila. Ti podaci biće korisni za efikasnije nalaženje narednih preseka. Ova tehnika zove se *tehnika pokretne prave*.

Zamislimo vertikalnu pravu koja prelazi ravan sleva udesno. Da bismo ostvarili efekat prebrisavanja, sortiramo sve krajeve duži prema njihovim $x$-koordinatama. Dve krajnje tačke vertikalne duži imaju iste $x$-koordinate, pa se registruje samo jedna $x$-koordinata. Za horizontalne duži moraju se koristiti oba kraja. Posle sortiranja krajeva, duži se razmatraju jedna po jedna utvrđenim redosledom. Kao i obično pri induktivnom pristupu, pretpostavljamo da smo pronašli presečne tačke prethodnih duži, i da smo obezbedili neke dopunske informacije, pa

sada pokušavamo da obradimo sledeću duž i da unesemo neophodne dopune informacija. Prema tome struktura algoritma je u osnovi sledeća. Razmatramo krajeve duži jedan po jedan, sleva udesno. Koristimo informacije prikupljene do sada (nismo ih još specificirali) da obradimo kraj duži, pronađemo preseke u kojima ona učestvuje, i dopunjujemo informacije da bismo ih koristili pri sledećem nailasku na neki kraj duži. Osnovni problem je definisanje informacija koje treba prikupljati. Pokušajmo da pokrenemo algoritam da bismo otkrili koje su to informacije potrebne.

Prirodno je u induktivnu hipotezu uključiti poznavanje svih presečnih tačaka duži koje se nalaze levo od trenutnog položaja pokretne prave. Da li je bolje proveravati preseke kad se razmatra horizontalna ili vertikalna duž? Kad razmatramo vertikalnu duž, horizontalne duži koje je mogu seći još uvek se razmatraju (pošto nije dostignut njihov desni kraj). S druge strane, kad posmatramo bilo levi, bilo desni kraj horizontalne duži, mi ili nismo naišli na vertikalne duži koje je seku, ili smo ih već zaboravili. Dakle, bolje je preseke brojati prilikom nailaska na vertikalnu duž. Pretpostavimo da je pokretna prava trenutno preklopila vertikalnu duž $L$ (videti sliku 12). Kakve informacije su potrebne da se pronađu svi preseci u kojima učestvuje duž $L$? Pošto se pretpostavlja da su svi preseci levo od trenutnog položaja pokretne prave već poznati, nema potrebe razmatrati horizontalnu duž ako je njen desni kraj levo od pokretne prave. Prema tome, treba razmatrati samo one horizontalne duži čiji su levi krajevi levo, a desni krajevi desno od trenutnog položaja pokretne prave (na slici 12 takvih duži ima šest). Potrebno je čuvati listu takvih horizontalnih duži (odnosno njihovih $y$-koordinata). Kad se naiđe na vertikalnu duž $L$, potrebno je proveriti da li se ona seče sa tim horizontalnim dužima. Važno je primetiti da za nalaženje preseka sa $L$ ovde $x$-koordinate krajeva duži nisu od značaja. Mi već znamo da horizontalne duži iz liste imaju $x$-koordinate koje "pokrivaju" $x$-koordinatu duži $L$. Potrebno je proveriti samo $y$-koordinate horizontalnih duži iz liste, da bi se proverilo da li su one obuhvaćene opsegom $y$-koordinata duži $L$. Sada smo spremni da formulišemo induktivnu hipotezu.

**Induktivna hipoteza:** Neka je zadata lista od prvih $k$ sortiranih $x$-koordinata krajeva duži kao što je opisano, pri čemu je $x_k$ najveća od tih $x$-koordinata. Umemo da pronađemo sve preseke duži koji su levo od $x_k$, i da eliminišemo sve horizontalne duži koje su levo od $x_k$.

Za horizontalne duži koje se još uvek razmatraju reći ćemo da su kandidati (to su horizontalne duži čiji su levi krajevi levo, a desni krajevi desno od tekućeg položaja pokretne prave). Formiraćemo i održavati strukturu podataka koja sadrži skup kandidata. Odložićemo za trenutak analizu realizacije ove strukture podataka.

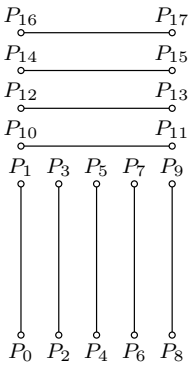
Bazni slučaj za navedenu induktivnu hipotezu je jednostavan. Da bismo je proširili, potrebno je da obradimo $(k+1)$-i kraj duži. Postoje tri mogućnosti.

1. $(k+1)$-i kraj duži je desni kraj horizontalne duži; duž se tada prosto eliminiše iz spiska kandidata. Kao što je rečeno, preseci se pronalaze pri

razmatranju vertikalnih duži, pa se ni jedan od preseka ne gubi eliminacijom horizontalne duži. Ovaj korak dakle proširuje induktivnu hipotezu.

2. $(k+1)$-i kraj duži je levi kraj horizontalne duži; duž se tada dodaje u spisak kandidata. Pošto desni kraj duži nije dostignut, duž se ne sme još eliminisati, pa je i u ovom slučaju induktivna hipoteza proširena na ispravan način.
3. $(k+1)$-i kraj duži je vertikalna duž. Ovo je osnovni deo algoritma. Preseci sa ovom vertikalnom duži mogu se pronaći upoređivanjem $y$-koordinata svih horizontalnih duži iz skupa kandidata sa $y$-koordinatama krajeva vertikalne duži.

Algoritam je sada kompletan. Broj upoređivanja će obično biti mnogo manji od $mn$, međutim, u najgorem slučaju ovaj algoritam ipak zahteva $O(mn)$ upoređivanja, čak i kad je broj preseka mali. Ako se, na primer, sve horizontalne duži prostiru sleva udesno ("celom širinom"), onda se mora proveriti presek svake vertikalne duži sa svim horizontalnim dužima, što implicira složenost $O(mn)$. Ovaj najgori slučaj pojavljuje se čak i ako nijedna vertikalna duž ne seče nijednu horizontalnu duž (slika 13).

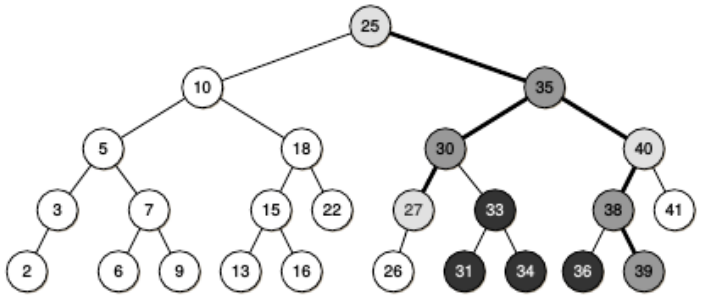



Slika 13: Slučaj kada nema preseka duži, a broj poređenja osnovnog algoritma je $O(mn)$.

Da bi se algoritam usavršio, potrebno je smanjiti broj upoređivanja $y$-koordinata vertikalne duži sa $y$-koordinatama horizontalnih duži u skupu kandidata. Neka su $y$-koordinate vertikalne duži koja se trenutno razmatra $y_D$ i $y_G$, i neka su $y$-koordinate horizontalnih duži iz skupa kandidata $y_0, y_1, \ldots, y_{r-1}$. Pretpostavimo da su horizontalne duži u skupu kandidata zadate sortirano prema $y$-koordinatama (odnosno niz $y_0, y_1, \ldots, y_{r-1}$ je rastući). Horizontalne duži iz skupa kandidata koje se seku sa vertikalnom duži mogu se pronaći izvođenjem dve binarne pretrage, jedne za $y_D$, a druge za $y_G$. Neka je $y_i < y_D \leq y_{i+1} \leq y_j \leq y_G < y_{j+1}$. Vertikalnu duž seku horizontalne duži sa koordinatama $y_{i+1}, y_{i+2}, \ldots, y_j$, i samo one. Može se takođe izvršiti jedna binarna pretraga za $y_D$, a zatim prolaziti $y$-koordinate, dok se ne dođe do vrednosti $y_{j+1}$ veće

od $y_G$. Iako je polazni problem dvodimenzionalan, nalaženje $y_{i+1}, \ldots y_j$ je jednodimenzionalni problem. Traženje brojeva u jednodimenzionalnom opsegu (u ovom slučaju od $y_D$ do $y_G$) zove se *jednodimenzionalna pretraga opsega*. Ako su brojevi sortirani, onda je vreme izvršenja jednodimenzionalne pretrage opsega proporcionalno zbiru vremena traženja prvog preseka i broja pronađenih preseka. Naravno, ne možemo da priuštimo sebi sortiranje horizontalnih duži pri svakom nailasku na vertikalnu duž.

Prisetimo se još jednom zahteva. Potrebna je struktura podataka pogodna za čuvanje kandidata, koja dozvoljava umetanje novog elementa, brisanje elementa i efikasno izvršenje jednodimenzionalne pretrage opsega. Na sreću, postoji više struktura podataka — na primer, uravnoteženo drvo — koja omogućuju izvršenje umetanja, brisanja i traženja elemenata složenosti $O(\log n)$ po operaciji (gde je $n$ broj elemenata u strukturi podataka), i linearno pretraživanje za vreme proporcionalno broju pronađenih elemenata. Pretraga započinje traženjem $y_D$ i $y_G$ u drvetu koje sadrži kandidate $y_0, \ldots, y_{r-1}$. Neka su putevi koji se pri tome prelaze polazeći od korena drveta označeni sa $P_D$, $P_G$, i neka je $v$ poslednji zajednički čvor za ta dva puta. Rezultat pretrage su ključevi kompletnih desnih podstabala čvorova sa puta $P_D$ (ispod $v$) u kojima put nastavlja levom granom, i (simetrično) kompletnih levih podstabala čvorova sa puta $P_G$ (ispod $v$) u kojima put nastavlja desnom granom. Ovim su obuhvaćeni svi čvorovi u unutrašnjim poddrvetima koja su ograđena putevima $P_D$ i $P_G$. Tokom obilaska, takođe se proveravaju i prijavljuju svi čvorovi na putevima $P_D$ i $P_G$ koji leže u datom intervalu.




Slika 14: Primer jednodimenzionalne pretrage opsega. Kandidati, njih 25, upisani su u uravnoteženo drvo, a ispisuju se kandidati iz intervala [29,39]. Poslednji zajednički čvor za dva puta je čvor $v$ sa vrednošću 35. Rezultat čine tamno sivi čvorovi (čvorovi na putevima $P_D$ i $P_G$ koji se nalaze u intervalu) i crni čvorovi (čvorovi u unutrašnjim poddrvetima ograničenim putevima $P_D$ i $P_G$).

Početno sortiranje prema $x$-koordinatama krajeva duži zahteva $O((m+n)\log(m+n))$ koraka. Pošto svako umetanje i brisanje zahteva $O(\log n)$ koraka, ukupno vreme za obradu horizontalnih duži je $O(n \log n)$. Obrada vertikalnih duži zahteva jednodimenzionalnu pretragu opsega, koja se može izvršiti za vreme

$O(\log n + r)$, gde je $r$ broj preseka ove vertikalne duži. Vremenska složenost algoritma je dakle $O((m+n)\log(m+n) + R)$, gde je $R$ ukupan broj preseka horizontalnih i verktikalnih duži.