

Konstrukcija i Analiza Algoritama

-- Iva Milovanović --
12/1/2020

I: Grafovi:

- Pretraga
- Topološko sortiranje
- Artikulacioni čvorovi i mostovi
- Najkraći putevi u grafu
- Minimalno povezujuće stablo
- Eulerov ciklus
- Komponente jake povezanosti

II: Algoritmi za obradu teksta:

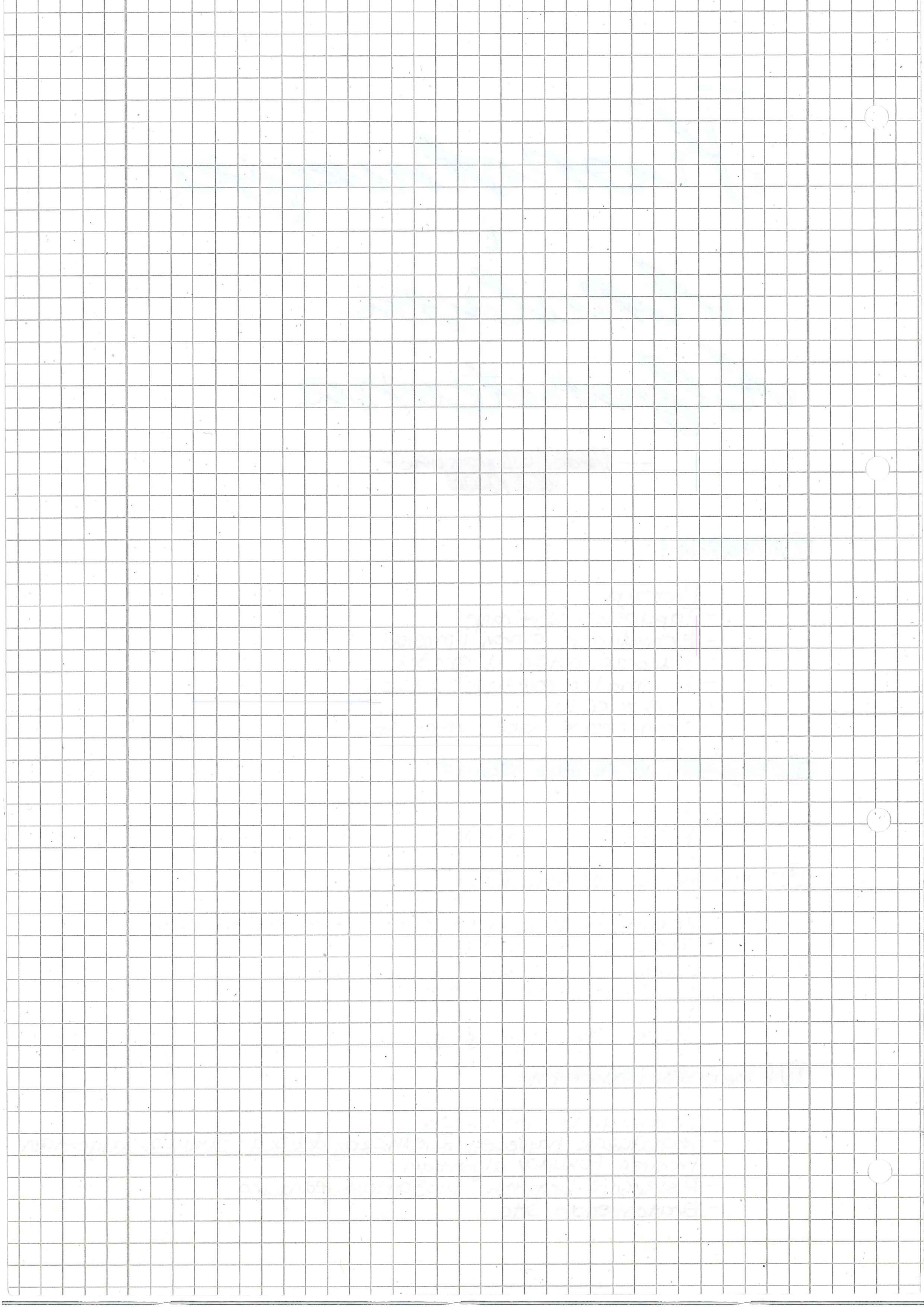
- Z-niz
- Manićerov algoritam

III: Geometrijski algoritmi:

- Vektori, skalarni i vektorski proizvod, orijentacija
- Tačka unutar ugla i trougla
- Ispitivanje konveksnosti mnogougla
- Konstrukcija prostog mnogougla
- Konstrukcija konveksnog omotaca
- Tačka unutar mnogougla

IV: Algebarski algoritmi:

- Kongruencija po modulu
- Pronalazak najvećeg zajedničkog delioca - Evklidov algoritam
- Proširen Evklidov algoritam
- Rešavanje linearnih Diofantovih jednačina
- Eratosthenovo Sito



GRAFOVI

- u opstem slučaju pričamo o neusmerenim grafovima, u suprotnom će biti naznačeno
- čvorove označavamo brojevima od 0 do $n-1$ za ukupno n čvorova
- postoje 2 reprezentacije grafova u računaru:

1. matrica povezanosti - binarna matrica $(n-1) \times (n-1)$ dimenzija gde su sa 1 označene pozicije sa indeksima gde su čvorovi povezani

2. lista suseda - vektor koji za svaki element / čvor čuva vektor čvorova sa kojima je povezan
 - redosled zavisi od unošenja
 - bolji način i više ćemo ga koristiti

* Učitavanje grafa:

```
int main() {
    int n, m;
    cin >> n >> m;

    Vector < Vector < int > graf(n);
    while (m--) {
        int a, b;
        cin >> a >> b;

        graf[a].push_back(b);
        graf[b].push_back(a);
    }
}
```

n - broj čvorova
 m - broj veza
 → krajevi grana

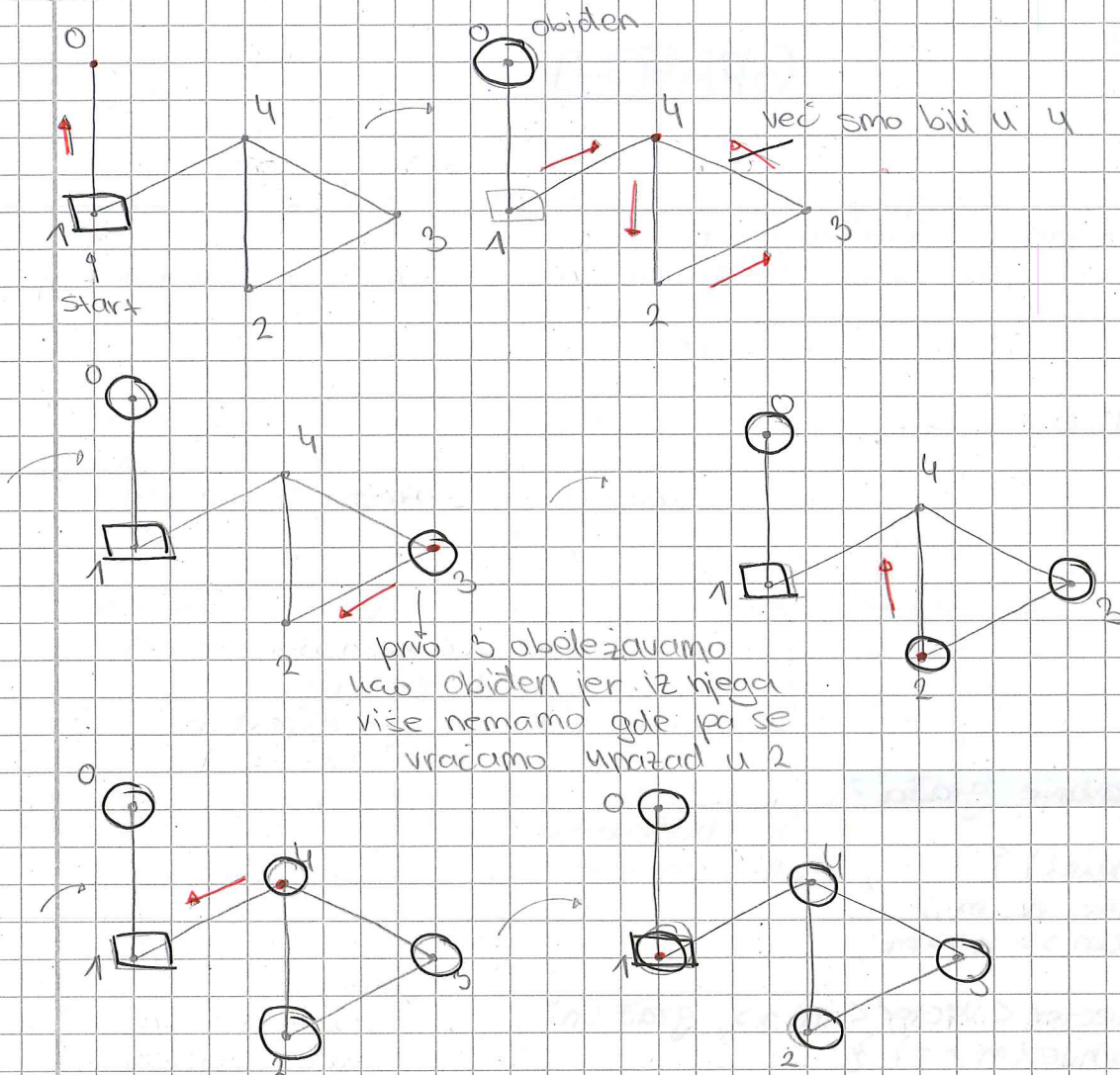
I Algoritmi pretrage grafa

- postoje 2 vrste pretrage:

1. pretraga u dubinu - DFS
2. pretraga u širinu - BFS

1. Pretraga u dubinu:

- zamislimo da se nalazimo u lavirintu
- kad god imamo novi novi hodnik (granu koja vodi do čvora koji nismo obišli) ućemo se tim hodnikom
- ovo radimo daleko gdje je to moguće
- zaustavljamo se i vraćamo nazad ukoliko smo došli u čvor gde nemamo dalje gde da idemo
 → slično kao backtracking sa AISP-a



* implementacija:

```
void dfs (int v, vector<bool> & obišten, vector<vector<int>> & graf) {
```

```
    obišten[v] = true;
```

```
    // cout << v << " "; → ako hoćemo ispis
```

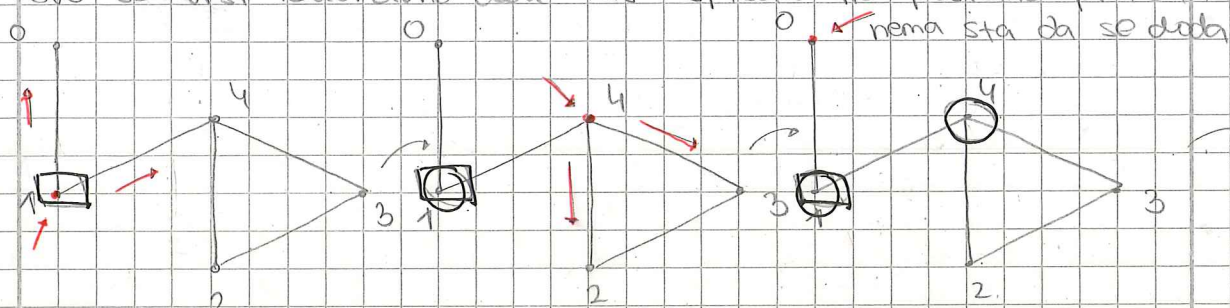
```
    for (auto w : graf[v])
```

```
        if (!obišten[w])
```

```
            dfs(w, obišten, graf); }
```

2. Pretraga u širinu

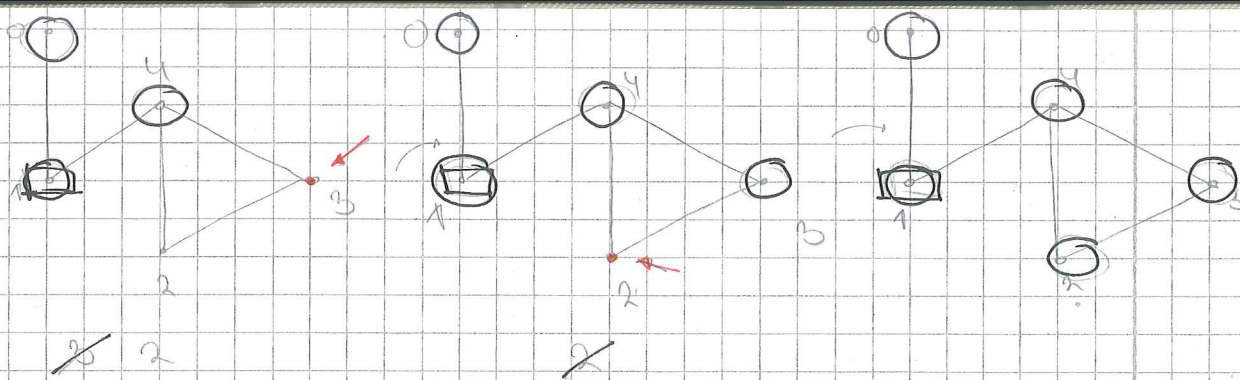
- gledamo do kojih čvorova možemo doći direktno u odnosu na trenutnu poziciju
- na "spisak" čvorova za pretragu dodajemo te čvorove
- po završenoj pretrazi čvora, izbacujemo ga sa "spiska"
- ovo se vrši rekurzivno dok naš "spisak" ne postane prazan



2 Spisak: ~~1~~ 4 0

~~1~~ 0 3 2

~~0~~ 3 2



* implementacija:

```
void bfs(int v, vector<vector<int>>& grafi) {
```

```
    vector<bool> obidjeni (grafi.size());
```

```
    queue<int> q; → lista gde svaddimo sa početka
```

```
    q.push(v);
```

```
    obidjeni[v] = true;
```

dok spisak ne postane prazan
vršimo pretragu

```
    while (!q.empty()) {
```

```
        int t = q.front();
```

```
        q.pop();
```

```
        // cout << t << " ";
```

```
        for (int w : grafi[t])
```

```
            if (!obidjeni[w]) {
```

```
                q.push(w);
```

```
                obidjeni[w] = true; } }
```

dodaj neobidjen čvor na
spisak i označi ga
kao obidjenog

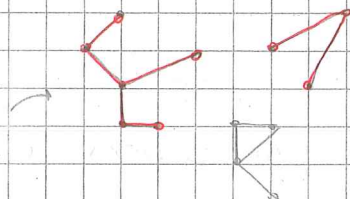
* Primene pretrage grafa:

1° Određivanje broja komponenti:

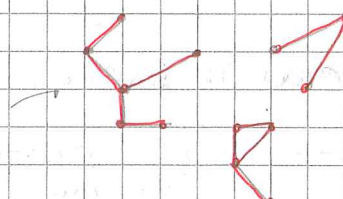
- ponavljanje pretrage za 1 čvor pronalazi i obrađuje sve čvorove koji se nalaze u istoj komponenti kao i početni, pa će ukupan broj komponenti predstavljati broj ponovljenih pretraga dok svi čvorovi ne budu obrađeni



1° dfs
brKomponenti++



2° dfs
brKomponenti++



3° dfs
brKomponenti++

```
int main() {
```

```
    -- učitavanje grafa --
```

```
    vector<bool> obiden (n, false);
```

```
    int brKomponenti = 0;
```

```
    for (int v = 0; v < n; v++)
```

```
        if (!obiden[v]) {
```

```
            dfs(v, obiden, grafi);
```

```
            brKomponenti++; } }
```


2. Određivanje veličine komponente:

- umesto da brojimo pozive dfs-a unutar main-a, ovde ćemo za poslednji element brojati koliko puta se dfs interno poziva

```
void dfs(int v, int& brojac, vector<bool>& obiden,
         vector<vector<int>>& graf) {
```

```
    obiden[v] = true;
    brojac++;
```

```
    for (auto w : graf[v])
        if (!obiden[w])
            dfs(v, brojac, obiden, graf);
}
```

```
int main() {
    -- učitavanje grafa --
    int x;
    cin >> x;
    int br = 0;
    vector<bool> obiden(n, false);
    dfs(x, br, obiden, graf);

    cout << br << "\n";
    return 0;
}
```

- ako ne želimo da radimo sa brojačem možemo konstitui povratnu vrednost 3-je uao brojac

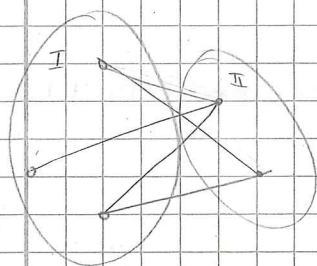
```
int dfs(int v, vector<bool>& obiden, int& brojac, vector<vector<int>>& graf) {
    obiden[v] = true;
    int brObidenih = 1;

    for (auto w : graf[v])
        if (!obiden[w])
            brObidenih += dfs(w, obiden, brojac, graf);

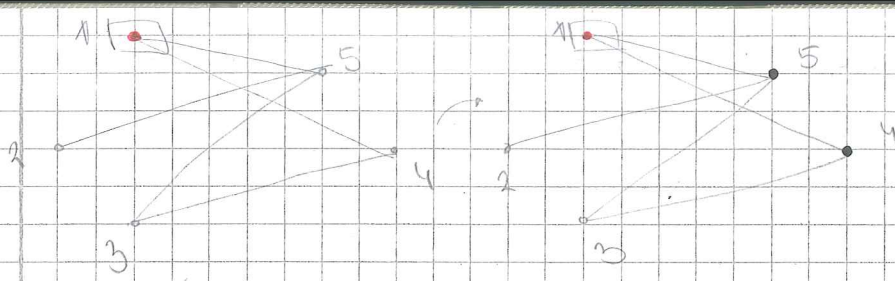
    return brObidenih;
}
```

3. Provera da li je graf bipartitan:

- bipartitan graf - graf čiji skup čvorova se može podeliti na dva podgrafa takva da čvorovi unutar njih nemaju nikakvih veza

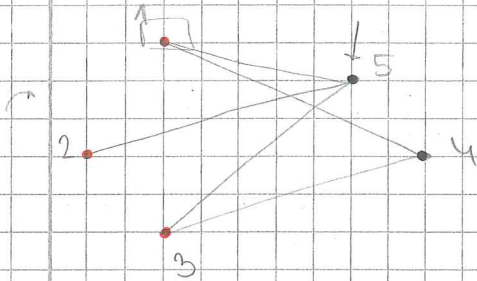


- provera da li je graf bipartitan se svodi na proveru suseda čvorova, tj. susedi će se razlikovati po boji



1 obelježavamo
crvenom

Susede od 1
Smo obelježili crnom



→ pri ostatku pretrage nećemo
dobiti da neki čvor mora biti
crveni i crni pa je dat
građ bipartitan

Susede od 5
Smo obelježili crvenom

- ako bi dobili situaciju kada bi isti čvor morao biti dve različite boje, tada dat građ nije bipartitan
- oznake u algoritmu:

- 0 - čvor nije obraden
- 1 - čvor je crven
- 2 - čvor je crn

boje se MORAJU
rediti naizmenično

→ suprotna (crvena) = plava
suprotna (plava) = crvena
 $\text{suprotna}(x) = 3 - x$

```
bool dfs(int v, int trenutnaBoja, vector<int>& boja,
        vector<vector<int>>& građ) {
```

```
    boja[v] = trenutnaBoja;
```

```
    for (auto w : građ[v]) {
        if (boja[w] == boja[v])
```

```
        return false;
```

```
        else if (boja[w] == 0 && !dfs(w, 3 - trenutnaBoja, boja, građ))
```

```
            return false; }
```

```
    return true; }
```

```
int main() {
```

```
    -- učitavanje građa --
```

postavljamo početnu boju

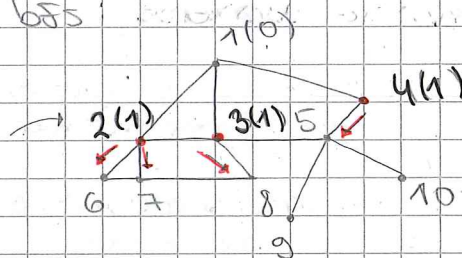
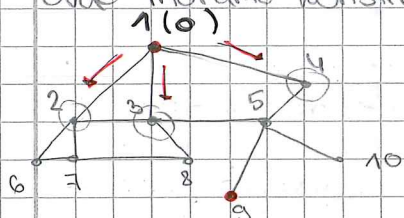
```
    vector<int> boja(n);
```

```
    bool bipartitan = dfs(0, 1, boja, građ);
```

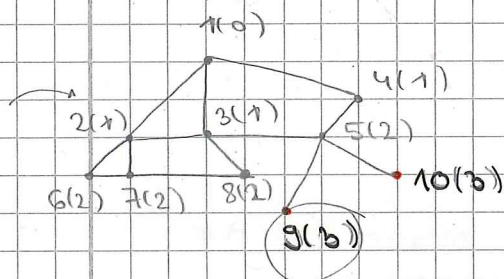
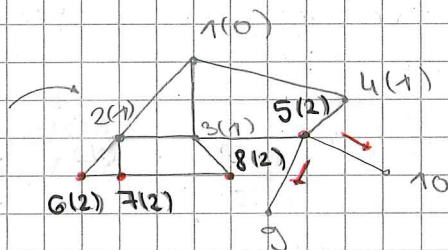
```
    ... }
```


4. Određivanje najkraćeg puta između neka dva čvora

- ovde moramo konstitui bfs:



- Zanima nas najkraća putanja od 1 do 9
- 0 - najkraća dužina od 1 do samog sebe je 0
- 2, 3, 4 se prvo obiđete



→ dobijamo da je najmanja udaljenost od 1 do 9 jednaka 3

- primetimo da ovim postupkom dobijamo udaljenosti od našeg čvora do SVIH čvorova u grafu, a ne samo do traženog
- 0 je sada rezervisana za udaljenost čvora od samog sebe, pa ćemo konstitui -1 za obeležje kada neki čvor nismo još obradili

```
vector<int> bfs(int v, vector<vector<int>>& graf) {
```

```
    vector<int> dist(graf.size(), -1);
    queue<int> q;
    q.push(v);
    dist[v] = 0;
```

```
    while (!q.empty()) {
        int t = q.front();
        q.pop();
```

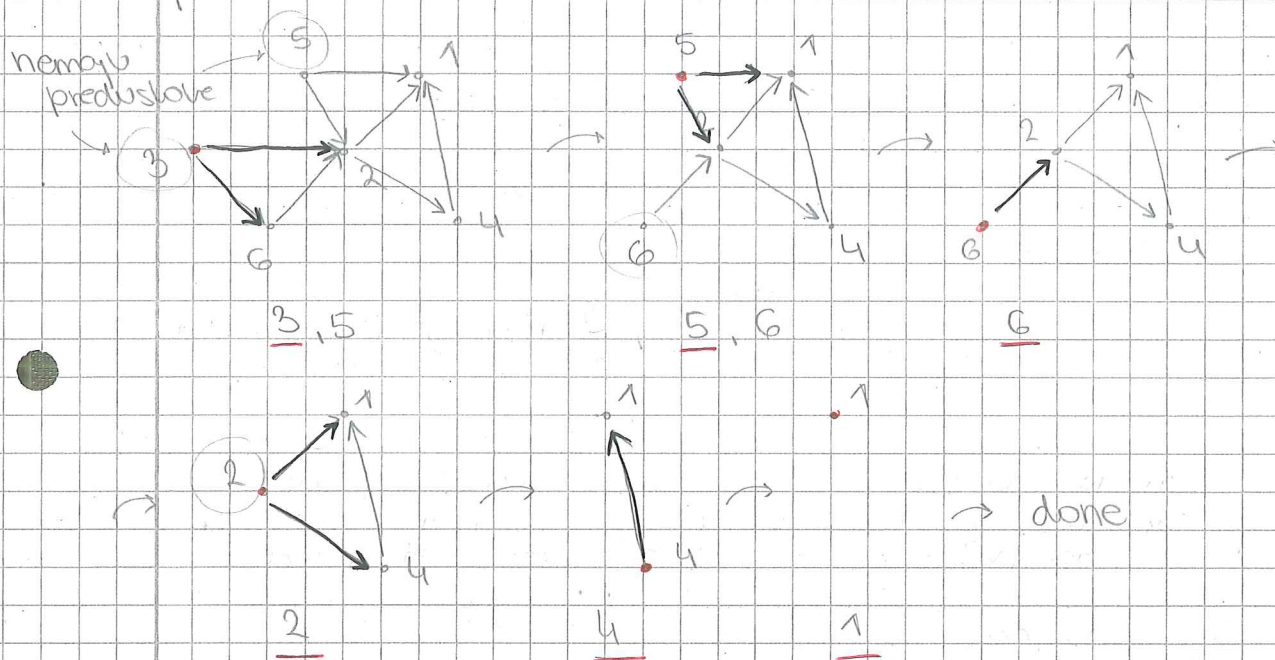
```
        for (int u : graf[t])
            if (dist[u] == -1) {
                q.push(u);
                dist[u] = dist[t] + 1;
            }
```

```
    return dist; }
```


ČAS 2:

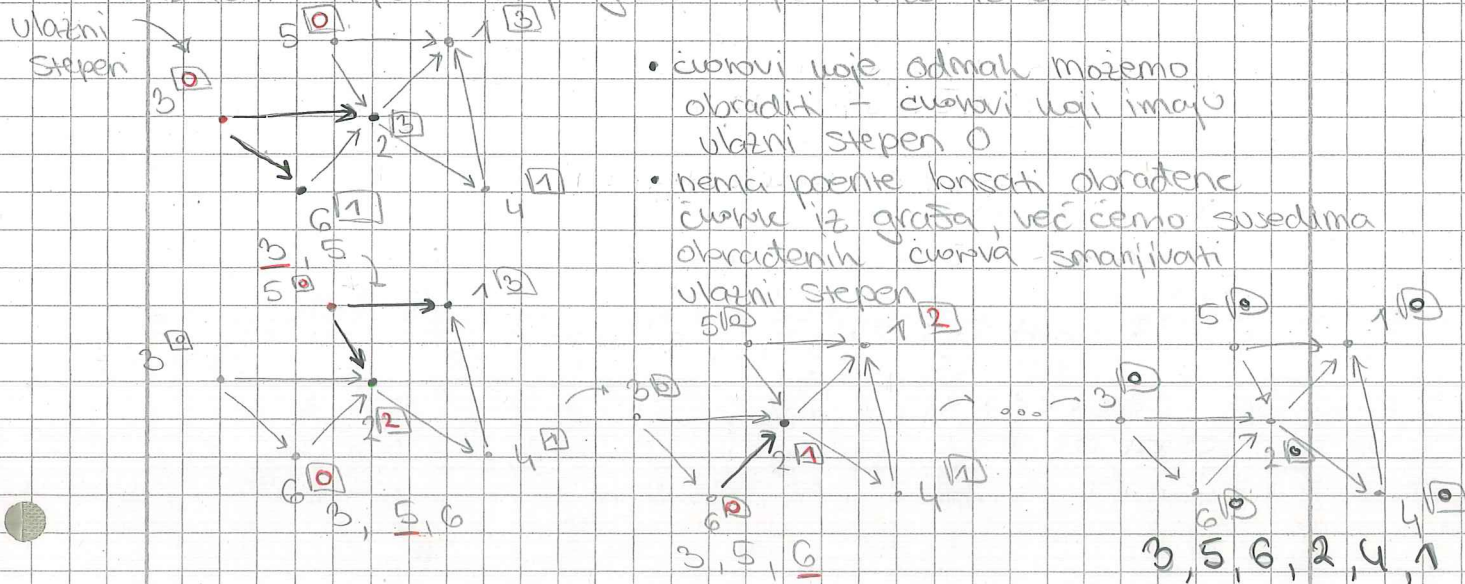
II Topološko sortiranje (uređenje):

- ovde se tipično radi o usmerenim grafovima
- posmatramo kao zadatak iz matematike gde nam je rešenje trenutnog zadatka potrebno rešenje prethodnog
- glavno pitanje - kojim redom treba raditi zadatke (obradivati čvorove) da bi došli do krajnjeg rešenja
- prvi korak - pronaći zadatke (čvorove) koji nemaju nikakav preduslov



- dobijamo topološko sortiranje: 3, 5, 6, 2, 4, 1
- može se desiti i da imamo više validnih topoloških sortiranja za neki graf, npr. ovde smo mogli početi i sa 5
- usmeren graf nema validno topološko uređenje → graf ima ciklus
→ pomoću topološkog sortiranja možemo dobiti informaciju o tome da li je graf acikličan ili acikličan

- ulazni stepen - broj grana koje ulaze u čvor (ovde bitan)
- izlazni stepen - broj grana koje izlaze iz čvora



⊗ implementacija:

```
vector<int> topološko (vector<vector<int>>& grafi) {
```

```
    int n = grafi.size();
    vector<int> indeg(n, 0);
```

→ ulazni stepeni:

-- inicijalizacija ulaznih stepena --

```
    for (int u=0; u<n; u++)
        for (int v: grafi[u])
            indeg[v]++;
```

za svaki čvor (u) uzimamo listu suseda prema kojim je usmeren (v) i povećavamo njihov ulazni stepen za 1

-- ubacivanje čvorova čiji je početni stepen 0 u vektor Q --

```
    vector<int> Q;
    for (int u=0; u<n; u++)
        if (indeg[u] == 0)
            Q.push_back(u);
```

-- obrada ostalih čvorova --

```
    for (int i=0; i<Q.size(); i++) {
        int u = Q[i];
        for (int v: grafi[u]) {
```

uzimaj naredni čvor duple
god postoji čvor sa
ulaznim stepenom 0

nakon obrade čvora u
svim njegovim susedima
smanji ulazni stepen za 1

```
            if (indeg[v] == 0)
                Q.push_back(v); }
```

ako se nekom čvoru
ulazni stepen smanji na
0 dodaj ga na listu za
obradu

```
    return Q; }
```

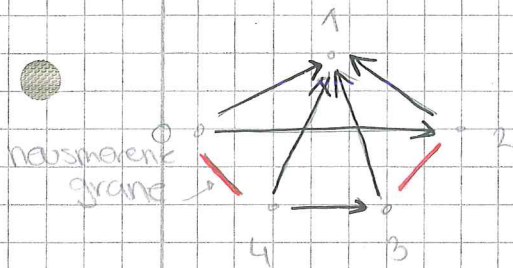
• u slučaju kada imamo ciklus, vektor Q će nam imati manje elemenata od n

```
int main() {
    int n, m; cin >> n >> m;
    vector<vector<int>> grafi(n);
    while (m--) {
        int a, b;
        cin >> a >> b;
        grafi[a].push_back(b); }
```

posto je grafi usmeren
ne radimo za čvor b
push_back(a)

```
    vector<int> top = topološko(grafi);
    if (top.size() < n)
        cout << "Grafi ima ciklus\n";
    else {
        for (int u: top)
            cout << u << " ";
        cout << '\n'; }
    return 0; }
```

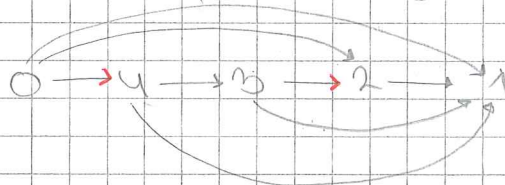

* primena - određivanje smera neusmerenih granama grafa tako da graf ostane acikličan:



- poznato je da je graf u startu acikličan, pa ćemo prvo odrediti njegovo topološko uređenje:

0, 4, 3, 2, 1

- posmatrajmo sada grane:



- grane idu sleva na desno →
- zaključujemo da neusmerene grane moramo usmeriti sleva na desno u odnosu na novo topološko uređenje, tj. tako da nemamo povratne grane
→ ovo nije jedino rešenje npr. možemo imati i granu 4 → 0, ali smer → garantuje acikličnost

- redosled topološkog uređenja odredit ćemo pomoću indeksa

top					index				
0	4	3	2	1	0	4	3	2	1
0	1	2	3	4	0	1	2	3	4

- Indekse ćemo pamtiti u posebnom vektoru radi jednostavnosti

// m1 - broj usmerenih grana
// m2 - broj neusmerenih grana

```
int main() {
    int n, m1, m2;
    cin >> n >> m1 >> m2;
    vector<vector<int>>> graf;
    while (m1--) { učitavanje usmerenog dela grafa }
```

```
    vector<int> top = topolosko (graf);
```

```
    // indeksiranje elemenata topološkog niza
```

```
    vector<int> index(n);
    for (int i=0; i<n; i++)
        index[top[i]] = i;
```

```
    while (m2--) {
        int a, b;
        cin >> a >> b;
```

```
        if (index[a] < index[b])
            cout << a << " " << b << "\n";
        else
            cout << b << " " << a << "\n";
    }
```

```
    return 0; }
```

[m - broj usmerenih grana
max broj neusmerenih:
 $\binom{n}{2} - m$]

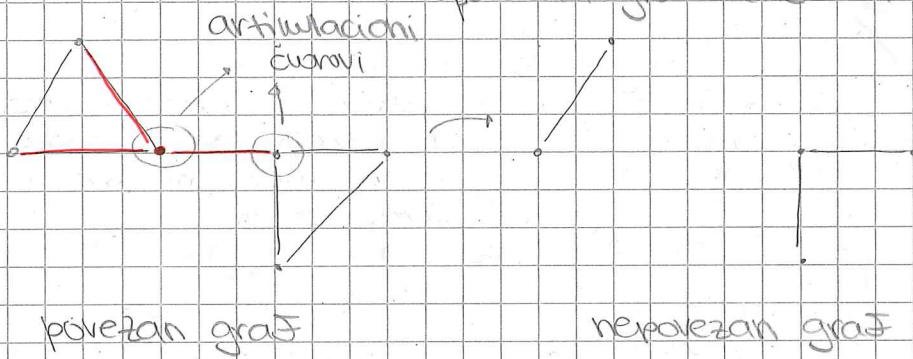
↑ broj mogućih grana između n čvorova
 $\binom{n}{2} = \frac{n(n-1)}{2}$

- grana između dva čvora može postojati samo u jednom smeru

III Articulacioni čvorovi i mostovi

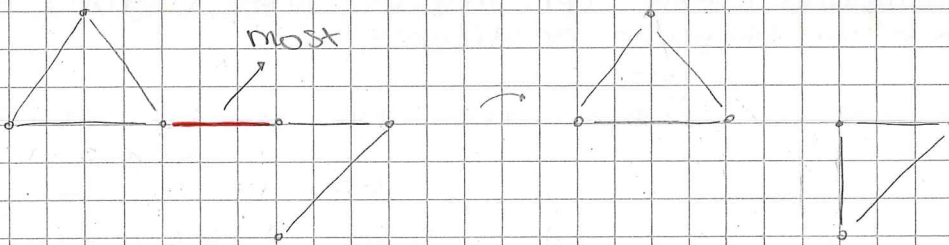
- píšeme o neusmerením grafu

⊗ Artikulacioni čvor - čvor čijim uklanjanjem dobijamo nepovezani graf (više od 1 komponente)



- korisno pri radu sa računarskim mrežama

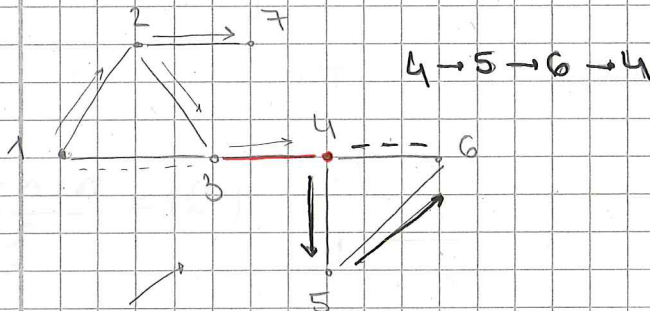
* Mostovi - grane između čvorova čijim uklanjanjem dobijamo nepovezani graf



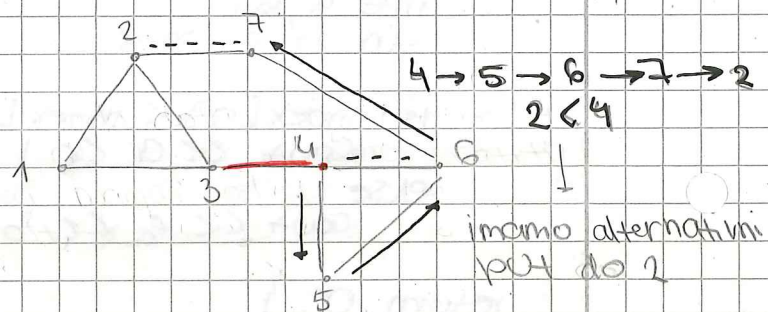
- izbacivanjem čvorova (redom) i proverom povezanosti dobijamo algoritam složenosti $O(n(n+m))$
- izbacivanjem grana - II - n - br. čvorova
dobijamo algoritam složenosti $O(m(n+m))$ m - br. grana
- želimo bolju složenost

1° Otuvivanje mostova:

- ako je grana između čvorova a i b most, nakon njenog uklanjanja a i b više nisu povezani, tj. ne postoji alternativni put između njih
- za svaku granu ćemo proveriti prethodno
- uostaćemo pretragu u dubinu (DFS)



- Ako je naša grana most nećemo moći da dobavimo čvor sa njene druge strane tj. iz 4, 5, 6. Se ne možemo vratiti unazad, 4 čvor sa numeracijom manjom od 4.



- u suprotnom, kada grana nije most, to je moguće

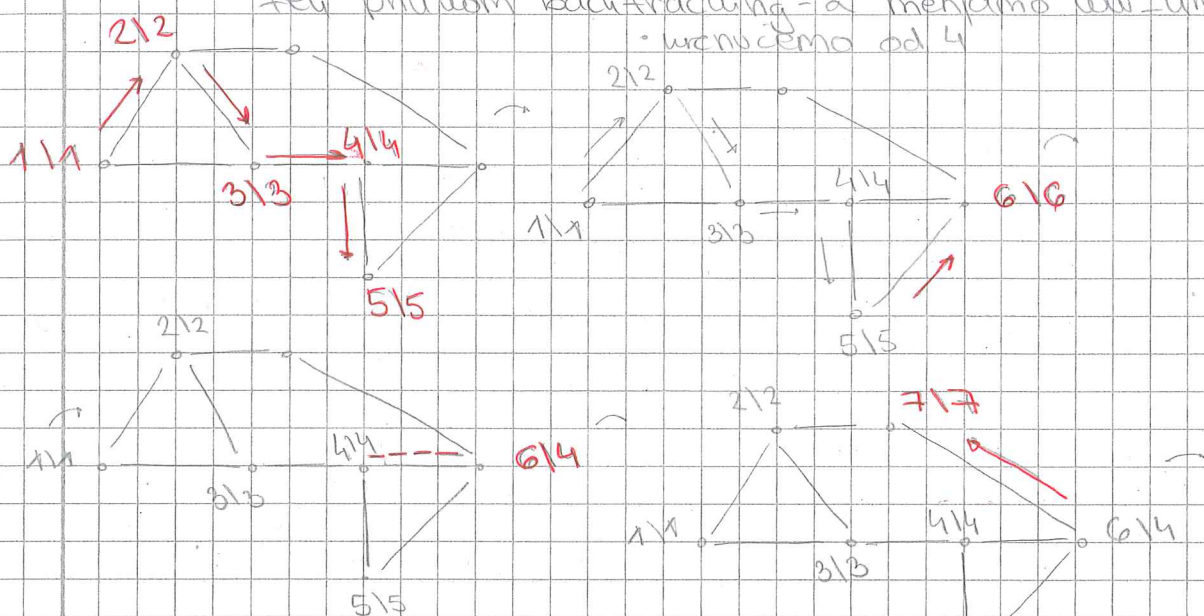
- problem - u ovom slučaju možemo da dosegamo počevši od nekog čvora

* Low-link vrednost - najmanja numeracija čvora do kog možemo da stignemo

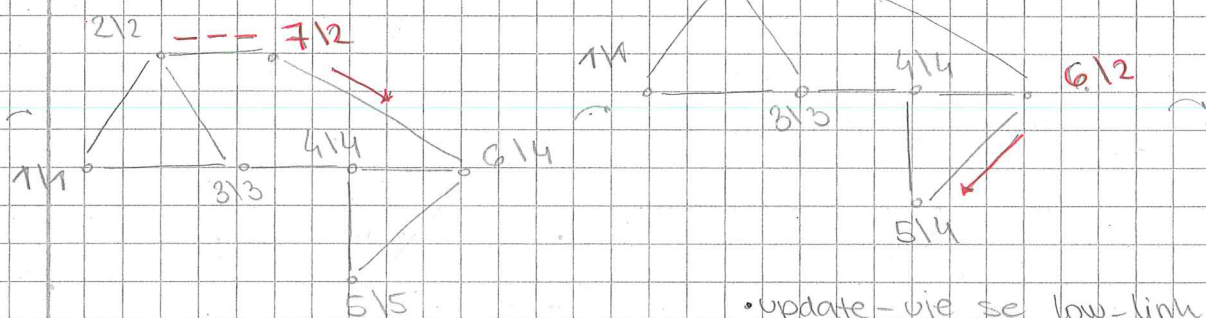
→ num - čvora / low-link-value

- low-link - inicijalno je jednaka numeraciji samog čvora
- BITNO - krećemo se na d3S i ne vraćamo se nazad, tek prilikom backtracking-a menjamo low-link

• vraćamo od 4

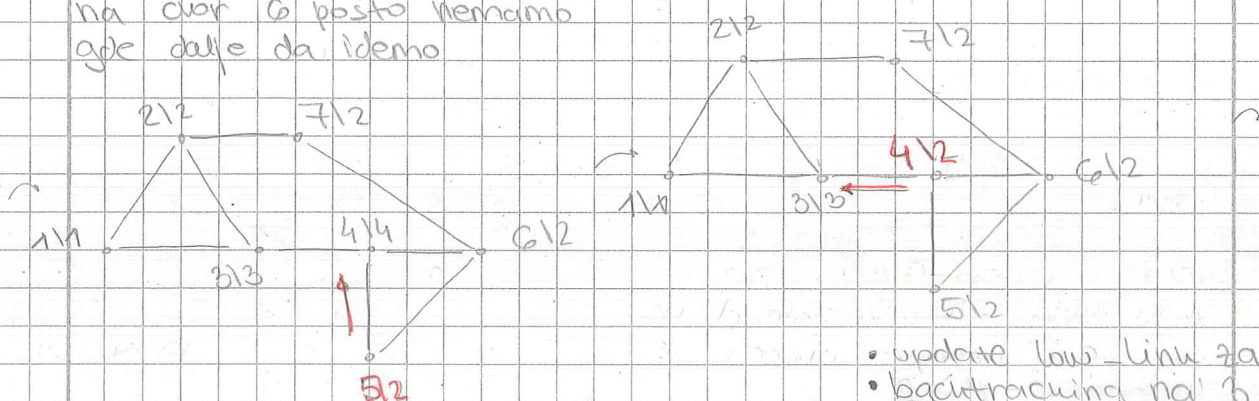


- našli smo na povratnu granu, tj. na čvor koji je već obišten, pa low-link za 6 postaje 4
- nakon toga nastavljamo pretragu



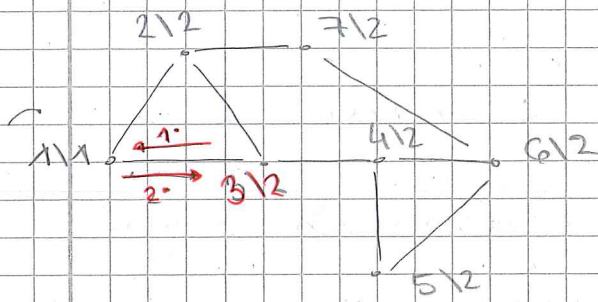
- isto kao i na pre, nailazimo na čvor koji je već obišten pa on postaje low-link ako je manji od trenutnog
- nakon toga radimo backtracking na čvor 6 posto nemamo gde dalje da idemo

- update - uje se low-link čvora 6
- njegove susede smo takodje obišli pa backtracu-ujemo na 5

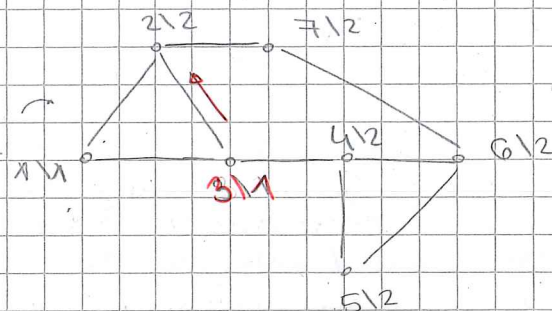


- update - uje se low-link za 5
- obrađene su grane od 5 - radimo b.t na 4

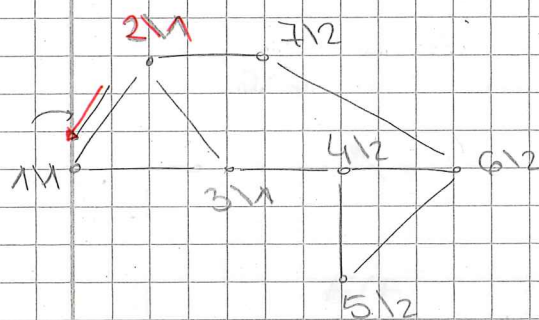
- update low-link za 4
- backtracking na 3



- misimo obišli sve susjede čvora 3 pa idemo u 1
- u 1 smo već obišli sve susjede pa ćemo se odmah vratiti u 3 backtracking-om



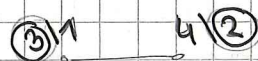
- update - uje se low_link čvora 3 pa, pošto nemamo više grana, vraćamo se u 2



- kada su sve low-link vrednosti ovako određene sve što treba urediti je proveriti da li je low-link čvora manja od numeracije parent čvora, tj. čvora od kog smo došli, ako jeste, nije u pitanju most

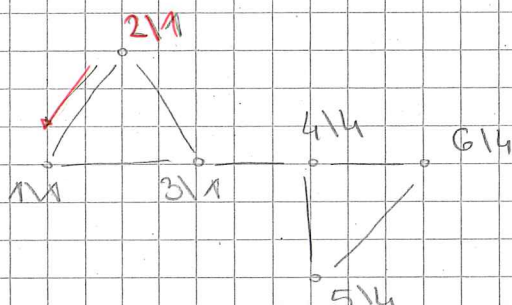
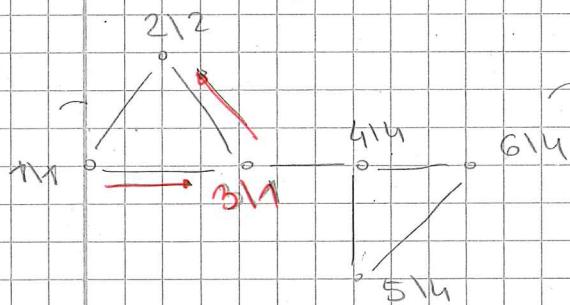
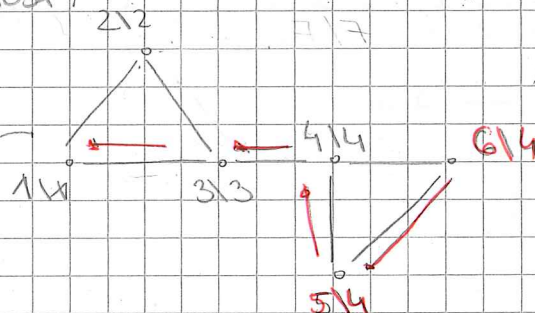
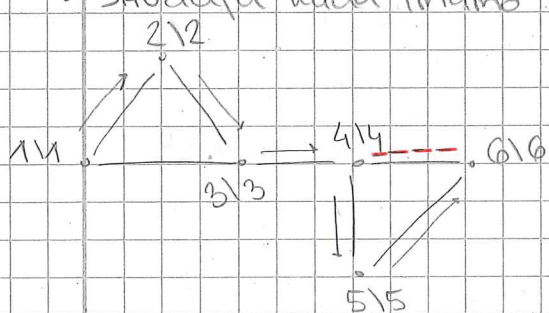
- update - uje se low_link od 2
- backtracking u 1
- završava se pretraga

- ovde za granu:

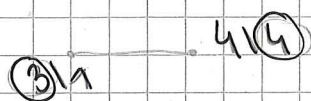


$2 < 3 \rightarrow$ grana 3-4 nije most

- situacija kada imamo most:



- za granu:



$4 > 3 \rightarrow$ grana 3-4 jeste most

- sličan je algoritam za određivanje artikulacionih čvorova, samo tada za prvi čvor treba imati u vidu da njegova low-link vrednost nikada ne može biti manja od njega samog (1/1)

⊗ implementacija:

- dfs će nam sada vraćati low-link vrednost čvora za koji se poziva (uod nas u)
- p - parent iz kojeg smo došli
- t - brojac - inicijalno 0 - uvećava se pri svakom ulasku u čvor
- num - vektor numeracije za svaki čvor
→ num[u] == 0 - čvor u nismo obišli

```
int dfs(int u, int p, int &t, vector<int> &num, vector<vector<int>> &graf){
```

```
    num[u] = ++t;  
    int low = num[u];
```

→ prvi čvor u koji uđemo
biće numerisan za 1

```
    for(int v: graf[u]){  
        if(num[v] == 0){
```

→ inicijalno low-link je
vrednost same numeracije

```
            int v_low = dfs(v, u, t, num, graf);
```

```
            if(num[u] < v_low) {  
                cout << u << " " << v << "n";
```

→ proveriti da li je grana
u-v most

```
                low = min(low, v_low);  
            }
```

```
        else if(v != p){
```

→ ako imamo povratnu granu
radimo update low-link-a

```
            low = min(low, num[v]);  
        }  
    }  
    npr. 4 --- 6 → min(6, 4)
```

→ ostala je samo situacija kada je u == p, tj. kada je
sused jednak roditelju, tj. vraćamo low-link trenutnog čvora

```
    return low; }
```

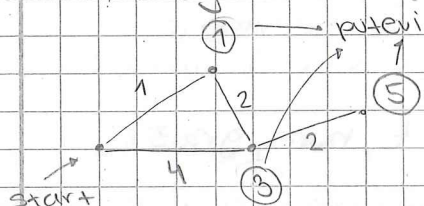

* CAS 3

IV Najkraći putevi u grafi:

- težinski grafi - grafi čije je svakoj grani dodeljena neka vrednost koja predstavlja dužinu/težinu te grane
- bfs - određuje najkraći put između dva čvora u grafu kada grafi nije težinski

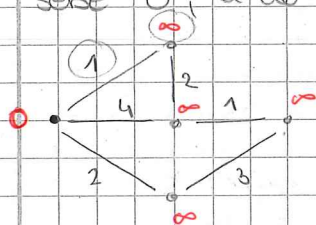
* Dijkstra / Dajkstrin algoritam - pronalazi najkraće dužine puteva između izabranog čvora i svih ostalih čvora u težinskom grafu

→ analogan bfs-u za ne-težinski grafi



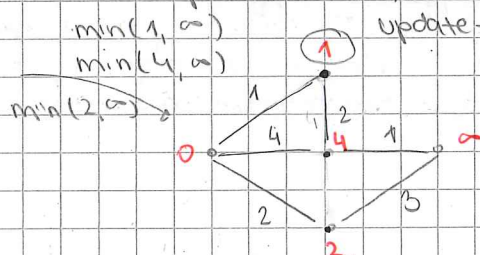
- ovaj algoritam će raditi i za usmerene i za neusmerene grafove

- početno postavljamo da je dužina od izabranog čvora do samog sebe 0, a do svih ostalih beskonačno



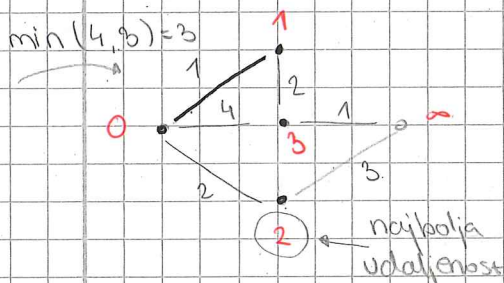
→ beležimo dužine do sada najkraće pronađenih puteva

- probavimo kroz sve susede izabranog čvora i update-ujemo najkraće puteve

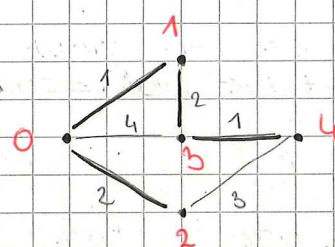
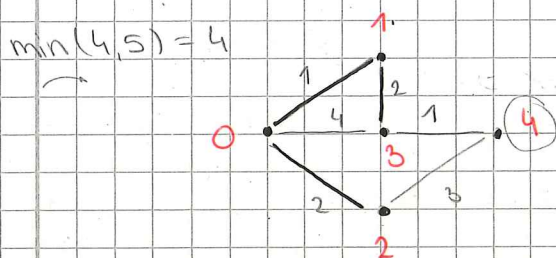
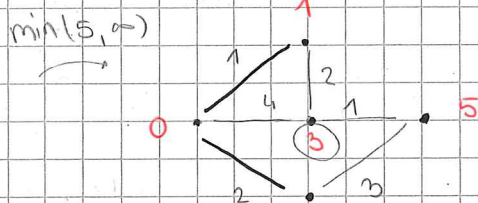


- kada smo obradili sve susede početnog čvora možemo reći da sigurno ovaj čvor koji ima najmanju od svih izračunatih udaljenosti u ostatku grafa jeste najbolji izbor za dalju pretragu suseda

→ nega više nećemo updateovati



- opet ažuriramo susede pa postupno ponavljamo



* implementacija:

- imamo niz koji nam za svaki čvor govori koja je njegova dosadashja udaljenost
→ posto nam treba minimum svih udaljenosti nedoradenih čvorova, umesto niza koristimo skup (set)
- grafi pravimo drugacije nego do sad jer osim informacije o povezanosti čvorova, sada moramo imati i težinu / dužinu grane, pa ćemo koristiti vektor parova
 - first - čvor do kog stižemo
 - second - dužina / težina grane

using edge = pair<int, int>;

- učitavanje grafa sada izgleda ovako:

```
vector<vector<edge>> graf(n);
while(m--) {
    int a, b, c;
    cin >> a >> b >> c;
    graf[a].push_back({b, c});
    graf[b].push_back({a, c});
}
```

- implementacija algoritma:

```
vector<int> dijekstra (int u, vector<vector<edge>> & graf) {
```

```
// inicijalizacija vektora udaljenosti svakog čvora od početnog
vector<int> dist(graf.size(), numeric_limits<int>::max());
dist[u] = 0;
```

- u C++ parovi se porede po default-u

$(a, b) < (c, d)$

- naš skup ćemo koristiti kao red

sa prioriteta gde se sortira po najkraćem putu, tj.:

$a < c \rightarrow \text{true}$
 $a > c \rightarrow \text{false}$

$(a, b) \rightarrow a$ je dužina

najkraćeg puta do čvora b

$a = c \ \&\& \ b < d \rightarrow \text{true}$
 $a = c \ \&\& \ b > d \rightarrow \text{false}$

- first - dužina / težina

- second - čvor do kog stižemo

- tehnici dužine iz skupa nećemo upisati koristiti, tj. skuzice nam samo za sortiranje unutar skupa

```
* [ set<pair<int, int>> red;
    red.insert({0, u});
    while (!red.empty()) {
        int v = red.begin() -> second;
        red.erase(red.begin());
```

uzimamo čvor sa min dužinom i izbacujemo ga iz skupa jer je obrađen

// posmatramo sve susede čvora do kog smo došli

```
for (edge e : graf[v]) {
    int sused = e.first;
    int duzina = e.second;
```


- ako je pronađena distanca manja od distance koja je do sada pronađena, onda iz swpa izbacujemo staru vrednost dužine, ažuriramo dužinu u dist vektoru i dodajemo u swp novu vrednost zbog sortiranja

```
if (dist[v] + dužina < dist[sused]) {
```

```
    red.erase({dist[sused], sused});
```

```
    dist[sused] = dist[v] + dužina;
```

```
    red.insert({dist[sused], sused}); } }
```

```
return dist; }
```

* Primena

- ako nam treba najkraci put od a do b preko c računamo dijustru od a do c i od c do b pa sabemo
 - ako treba zaobići/preskočiti neki čvor možemo ga jednostavno izbaciti iz grafa
 - ako nam treba najmanja udaljenost od više početnih čvorova (primer sa policijskim stanicama) samo sve početne čvorove upišemo u vektor dužina sa 0 i u swp sa 0 i poverujemo dijstruktnim algoritam
 - alternativno možemo dodati nov čvor u koji je na 0 udaljenost od početnih čvorova i izvršiti pretragu za njega
 - određivanje najkraceg puta - ako su dva puta iste dužine izabrati onaj sa manje grana
- kažemo da je udaljenost par vrednosti:

- First - dužina / težina

- Second - broj grana na putu

} udaljenost

using udaljenost = pair<int, int>;

int dijustra(int u, int k, vector<vector<edge>> & graf) {

vector<udaljenost> dist(graf.size(), numeric_limits<int>::max());

dist[u] = {0, 0};

set<pair<udaljenost, int>> red;

red.insert({{0, 0}, u});

{dužina, grane}

while(!red.empty()) {

int v = red.begin() -> second;

red.erase(red.begin());

for (edge e : graf[v]) {

int sused = e.first, dužina = e.second;

udaljenost nova_dužina = {dist[v].first + dužina, dist[v].second + 1};

if (nova_dužina < dist[sused]) {

red.erase({dist[sused], sused});

dist[sused] = nova_dužina;

red.insert({dist[sused], sused}); } }

return dist[k]; }

* CAS 4

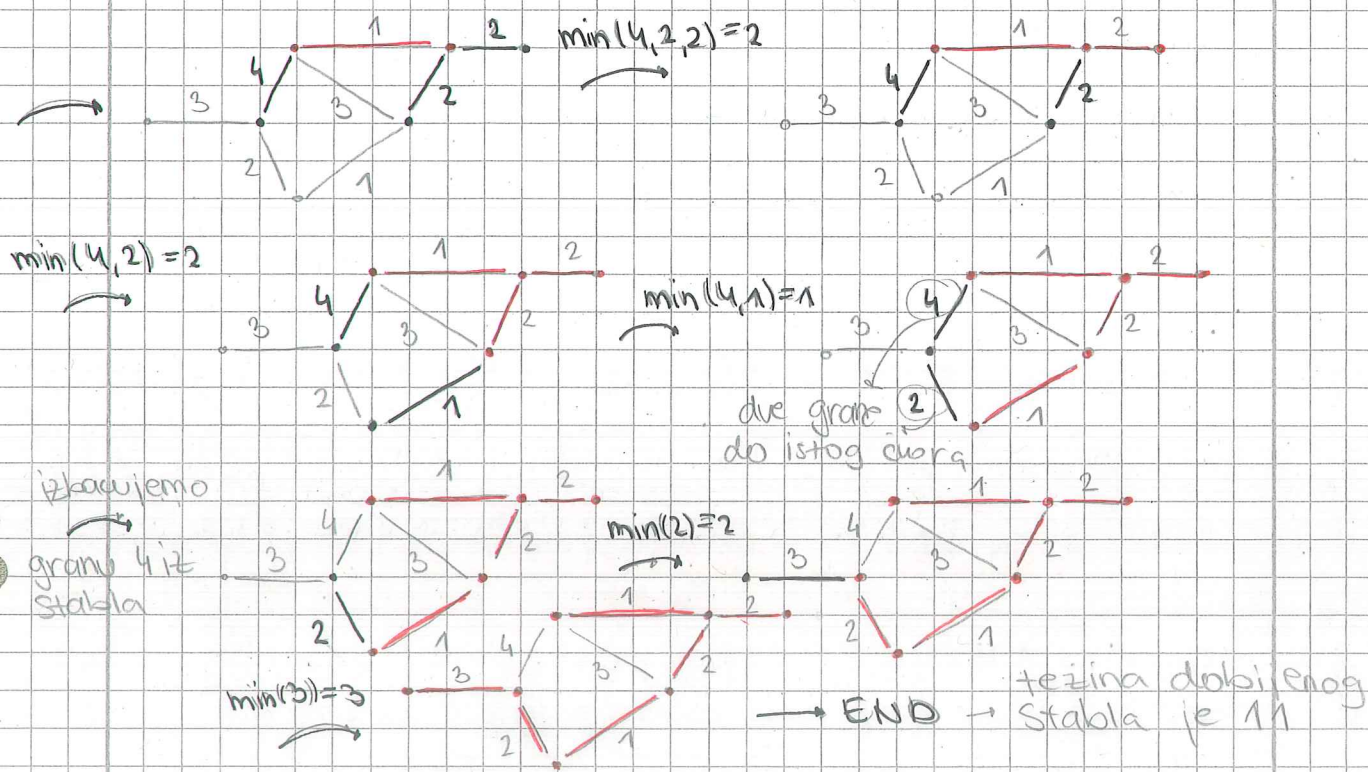
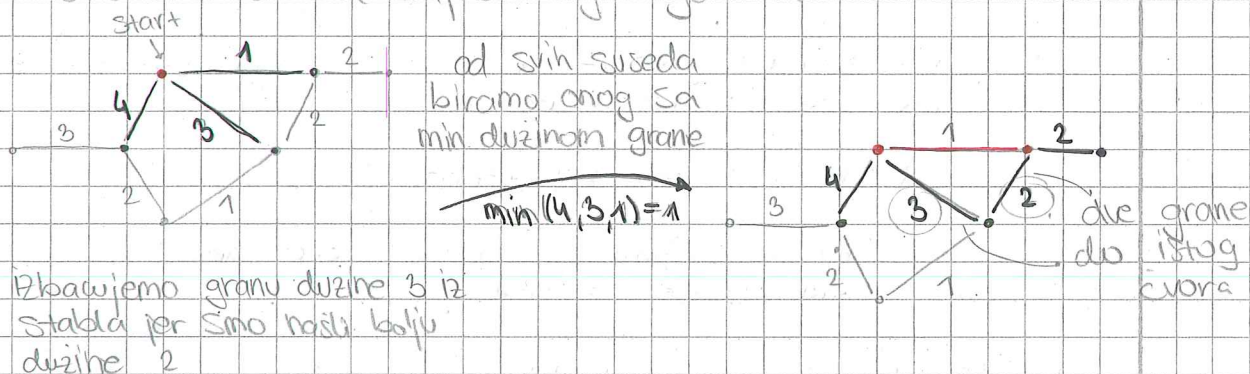
Minimalno povezujuće stablo

- minimalno stablo razapinjanja
- razapinjuće stablo - stablo koje se može dobiti kao podgrađ početnog grafa koje povezuje sve čvorove
→ da bi ovačno stablo postojalo građ MOGA biti povezan
- minimalno stablo razapinjanja - minimizuje zbir težina grana
→ krajnje rešenje će uvek biti stablo

* Primov algoritam

- počinjemo od proizvoljnog čvora za koji kažemo da sigurno pripada našem krajnjem stablu
- dolazimo se susede izabranog čvora
- pronalazimo najbolju direktnu vezu do čvorova
- pohlepan rad - u svakom koraku biramo čvor koji do sad nismo dodali u stablo i dodajemo njega i granu koja ga povezuje u stablo pri čemu biramo čvor koji ima najmanju težinu grane koja ga povezuje

→ slično kao kod dijekstrovog algoritma



* implementacija:

- bice nam potrebne 3 strukture:

- Set - skup koji nam sortira čvorove po težini
- Vector<bool> - parovi koji čvorovi su dodati u stablo
- Vector<edge> - Stablo

using edge = pair<int, int>; // {sused, težina}

vector<edge> prim(vector<vector<edge>> & graf){

// Stablo → parovi {roditelj, dužina grane}

// početno je svima roditelj -1 jer nisu u stablu, a dužina max

vector<edge> stablo(graf.size(), {-1, numeric_limits<int>::max()});
vector<bool> dodati(graf.size(), false);

// red → parovi {težina, sused} zbog sortiranja

set<pair<int, int>> red;

red.insert({0, 0}); // proizvoljni element grafa

while(!red.empty()){

int v = red.begin() -> second;

red.erase(red.begin());

dodati[v] = true;

pred ciklusa može biti
i do preida unutar stabla

for(auto grana : graf[v]){

int sused = grana.first;

int težina = grana.second;

moramo proveriti da li je
čvor već dodan jer ako jeste
može doći do ciklusa

if(!dodati[sused] && težina < stablo[sused].second){

red.erase({stablo[sused].second, sused});

stablo[sused] = težina;

red.insert({stablo[sused].second, sused}); } }

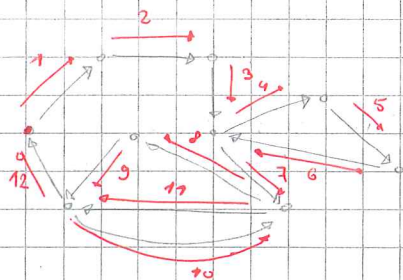
return stablo; }

VI Ojereni ciklus:

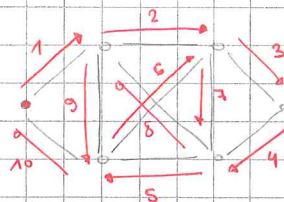
- Ojereni put u grafu

* Ojereni ciklus - ciklus koji sadrži sve grane grafa tačno jednom

1. Usmereni grafovi:



2. Neusmereni grafovi:



* Ojereni ciklus - vraćamo se u čvor iz koga smo krenuli

* Ojereni put - isto kao ciklus, ali ne moramo završiti u istom čvoru iz koga smo krenuli

- Usmeren graf ima Ojereni ciklus ako svaki čvor tog grafa ima jednake ulazne i izlazne stepene (broj ulaznih i izlaznih grana je jednak)
- Neusmereni graf ima Ojereni ciklus ako je stepen svakog njegovog čvora paran

- Kod Ojerenog puta početni i krajnji čvor moraju imati neparan stepen, tj. jednu granu za početni ulaz i jednu granu za krajnji izlaz



- početni čvor ima za 1 više izlaza nego ulaza
- krajnji čvor ima za 1 više ulaza nego izlaza

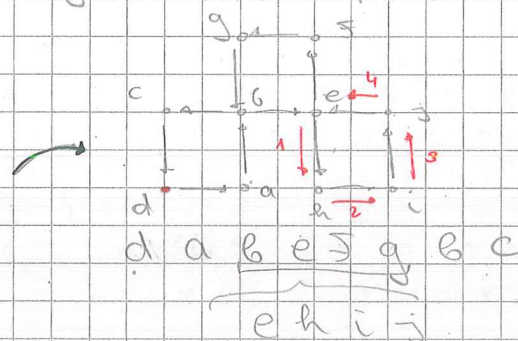
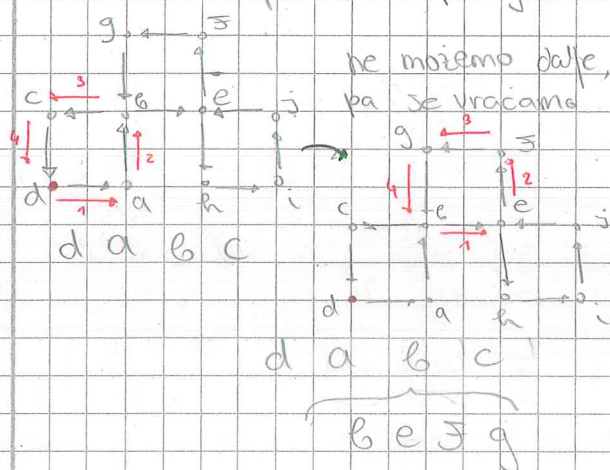
za usmeren graf

→ Graf ima Ojereni put ako postoje 2 čvora čiji je stepen neparan, a svima ostalima je stepen paran

→ Graf ima Ojereni put ako ima Ojereni ciklus

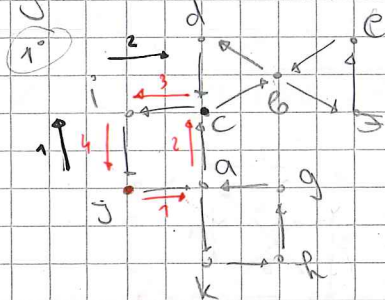
* implementacija za usmereni graf

- slično pretrazi u dubinu
- ućiemo se kroz graf daleko možemo, a kada ne možemo dalje bacit ćemo - vjero do prvog mesta gde možemo dalje



→ d a b e f g b c

- želimo da spojimo manje cikluse sa vecim, tj. nadovežemo
- od početnog cubra želimo da obidemo sve što može da se obide (sa ponavljanjem cubra i bez ponavljanja grana) i od toga da osformiramo ciklus



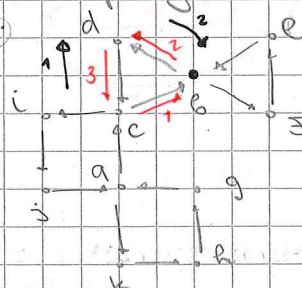
1°

- iz j idemo nekim putem i stizemo ponovo do j gde više nemamo gde pa zapisujemo j: j
- bacamo u i gde isto nemamo dalje gde pa zapisujemo i njega: i j
- bacamo u c, ali tu imamo dalje gde da idemo pa ga ne zapisujemo

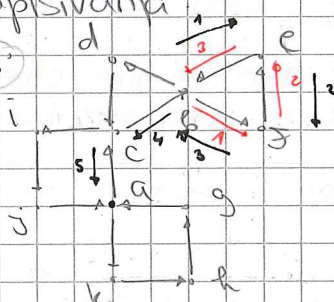
2°

- stizemo ponovo u c, ali sada nemamo dalje gde pa ga zapisujemo: c i j
- vraćamo se u d gde nemamo dalje pa ga zapisujemo: d c i j
- u b imamo gde, pa obilazimo dalje bez zapisivanja

2°



3°



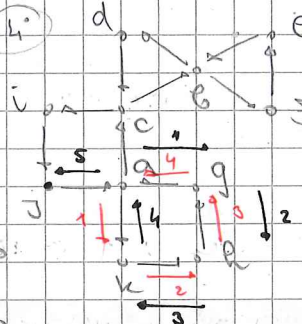
3°

- stizemo u b, nemamo dalje gde pa ga zapisujemo: b d c i j
- vraćamo se u e, nemamo gde pa ga zapisujemo: e b d c i j
- vraćamo se u j, nemamo gde pa ga zapisujemo: j e b d c i j
- vraćamo se u c, nemamo gde pa ga zapisujemo: c j e b d c i j
- vraćamo se u a, imamo gde

4°

- dolazimo opet u a, nemamo gde, zapisujemo: a c j e b d c i j
- redom se vraćamo kroz g, h, k i nemamo gde pa ih zapisujemo: k h g a c j e b d c i j
- vraćamo u a, nemamo gde, pa pišemo: a k h g a c j e b d c i j
- vraćamo u j, nemamo gde pa pišemo:

4°



j a k h g a c j e b d c i j

- ispisujemo na početku wren pa ćemo za ciklus koristiti den (deque)
- kad god pređemo preko neke grane, nju ćemo obrisati iz grafa
- kod rekurzije je najlakše koristiti pop-back pa ćemo susede obilaziti od kraja ka početku (sdesna ka levo)
- ovo se lako postiže rekurzijom, ali radi memorije i velikog broja grana grafova nju želimo da izbegnemo
- za ovaj problem ćemo koristiti stek (stack)


```
deque<int> opler (vector<vector<int>> grafi){
```

```
    deque<int> ciulus;  
    Stack<int> S;
```

kopiramo namerno per
brisemo grane grafi

```
    S.push(0); // pocetni element
```

```
    while(!S.empty()){  
        int u = S.top();
```

ako smo uvelonili sve grane
cvora dodajemo ga v ciulus
i uvelonimo sa steua

```
        if (grafi[u].empty()){  
            ciulus.push_front(u);  
            S.pop();
```

ako imamo gde dodajemo
narednog suseda na steu i
uvelonimo granu na njemu

```
        else {  
            S.push(grafi[u].back());  
            grafi[u].pop_back();
```

```
    }  
    return ciulus;
```

⊗ implementacija za neusmereni grafi

- posto je brisanje grana oide komplikovano, npr. obradjujemo
cvor a i obrisemo njegovu granu na b, ali i iz cvora b
moramo obrisati granu na a, koristiceemo matricu povezanosti

```
deque<int> opler (vector<vector<int>> grafi){
```

```
    vector<vector<bool>> susedi(grafi.size(), vector<bool>(grafi.size()));  
    for (int u=0; u<grafi.size(); u++)  
        for (int v : grafi[u])  
            susedi[u][v] = susedi[v][u] = true;
```

matrica
povezanosti

```
    deque<int> ciulus;  
    Stack<int> S;  
    S.push(0);
```

aktiviranje grafa

```
    while(!S.empty()){  
        int u = S.top();
```

dokle god ima granu iz cvora u i
dokle god za poslednjeg suseda i u
u matrici povezanosti stoji false
(obrisan je od strane suseda) brisi granu
iz cvora u

```
        while(!grafi[u].empty() && !susedi[u][grafi[u].back()])  
            grafi[u].pop_back();
```

```
        if (grafi[u].empty()){  
            ciulus.push_front(u);  
            S.pop();
```

```
        else {  
            int v = grafi[u].back();  
            S.push(v);  
            grafi[u].pop_back();  
            susedi[u][v] = susedi[v][u] = false;
```

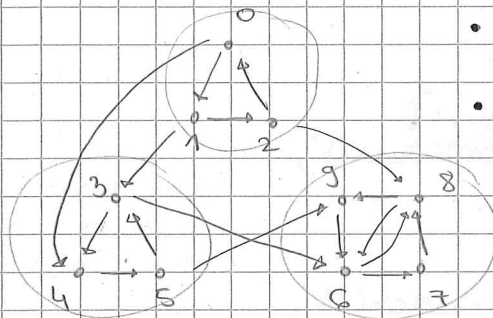
obrisi granu iz u u v
i update matricu naovo
bi uasnije v znao da
uveloni tu granu

```
    }  
    return ciulus;
```


ČAS 5:

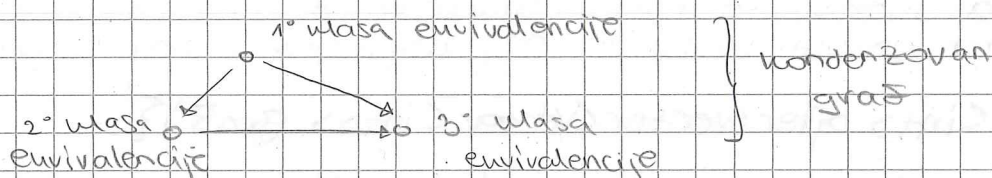
VI Komponente jake povezanosti:

- grafovi ade moraju biti usmereni
- zanimaju nas putevi / podgrafovi gde možemo da se vratimo u polazni čvor



- ako krenemo iz 0 možemo se vratiti do nula putem 0 1 2 0
- isto važi i za 3 4 5 3 i 6 7 8 9 6

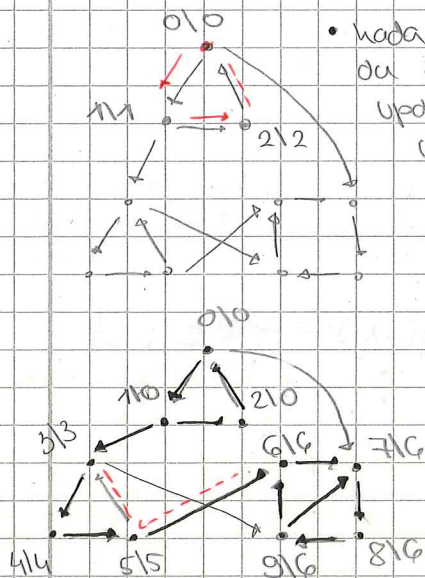
- ako posmatramo parove čvorova za koje važi mogućnost vraćanja u početni čvor dobijamo relaciju ekvivalencije, tj. postoji put od čvora A do čvora B i postoji put od čvora B do čvora A
- zbog izbora ekvivalencije među čvorovima možemo da particionisemo graf u klase ekvivalencije



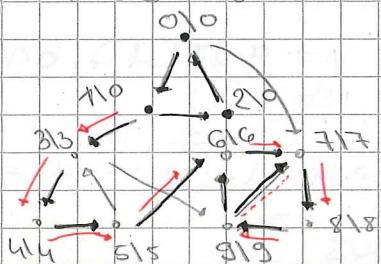
- ove klase ekvivalencije su komponente jake povezanosti
- ako graf ima bar 1 komponentu jake povezanosti, za taj graf možemo reći da je jako povezan
- kondenzacijom ćemo uvek dobiti acikličan graf

Implementacija:

- algoritam se oslanja na pretragu u dubinu i low link vrednosti
- pamtićemo numeraciju čvora i low-link vrednost čvora

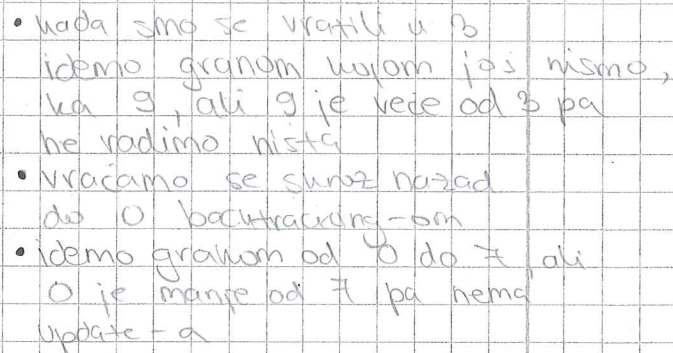


- kada stignemo do 2 jedini čvor u koji možemo da idemo je 0, ali u njemu smo već bili pa update-ujemo low-link i vraćamo se nazad u 1 gde takođe radimo update



- iz 9 prvo obilazimo 7 koji je već obišten pa dobijamo 9/7, ali onda idemo granom do 6 koji je takođe obišten, pa dobijamo 9/6
- nemamo gde dalje pa se vraćamo nazad i višimo update

- 5 je manje od 6 pa nema update-a
- iz 5 idemo u 3 koji je već obišten čvor pa radimo update i povratak nazad u 4 gde isto radimo update



- ```
int t = 0; → numeracijo za djs
vector<int> num;
vector<int> low;
stack<int> s;
vector<bool> obilazak;
```

low[u] = num[u] = ++t; → početna numeracija i low-link  
 s.push(u); } obeležavanje da trenutno radimo sa čvorom  
 dolazak[u] = true; } u, a na kraju će se svi čvorovi jedne komponente  
 nalaziti unutar istog, koji praznimo na kraju

else if (oblasak[V]) { → ako smo već bili u ovom nodu pripada  
low[u] = min(low[u], num[V]); } // trenutnom DFS oblasak

```

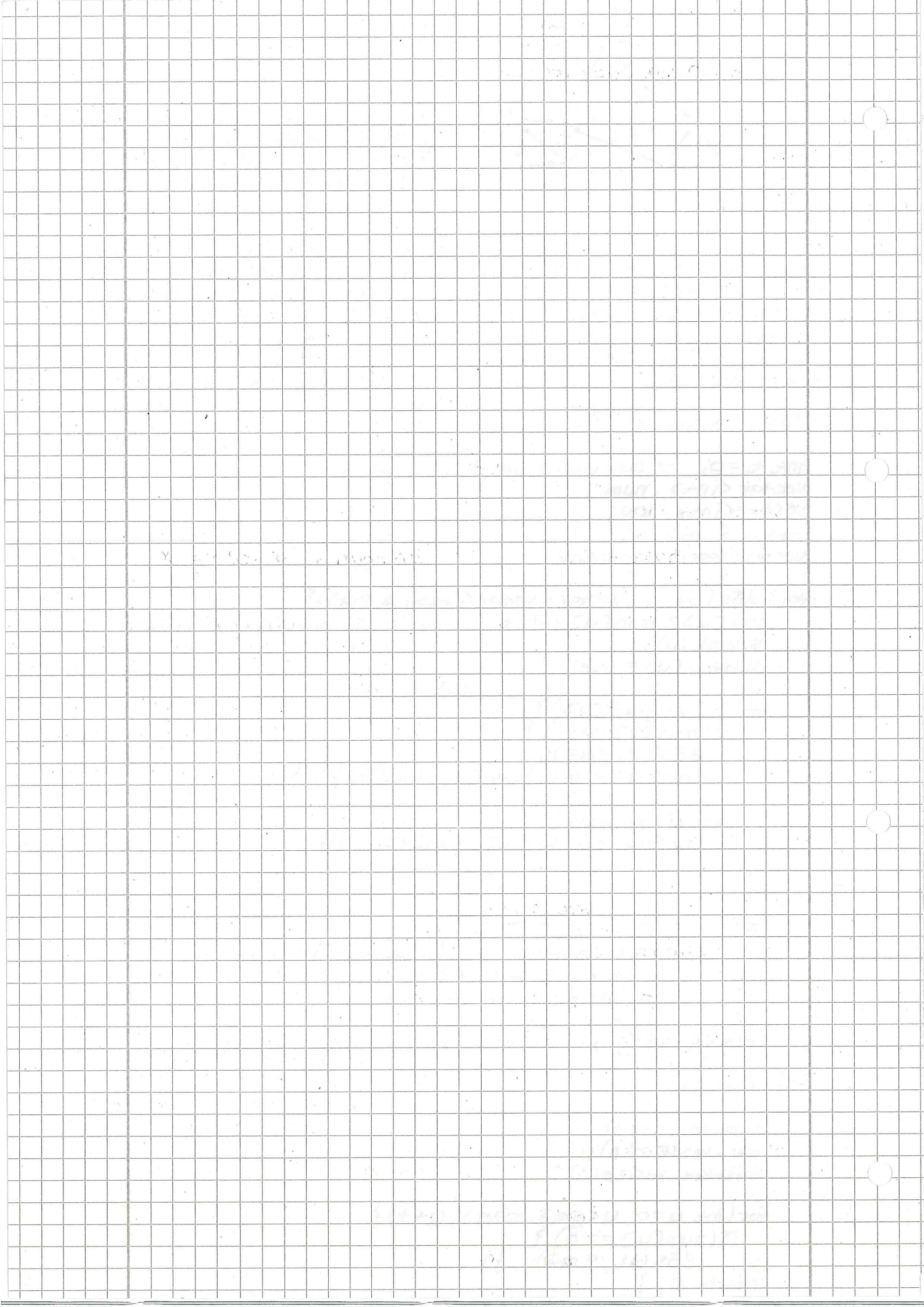
if (num[u] == low[u]) {
 while (!s.empty() && low[s.top()] == num[u]) {
 cout << s.top() << " ";
 oblaze[s.top()] = false;
 s.pop();
 }
 cout << "\n";
}

```

```
num.resize(n);
low.resize(n);
oblaqah.resize(n);
```

23







# I Algoritmi za obradu teksta

1° Z-niz:

S: a b a b a c

Z: 0 0 3 0 1 0

•  $a \rightarrow 0$

•  $b \neq a \rightarrow 0$

•  $a = a \rightarrow b = b \rightarrow a = a \rightarrow b \neq c \rightarrow 3$

•  $b \neq a \rightarrow 0$

•  $a = a \rightarrow c \neq b \rightarrow 1$

•  $c \neq a \rightarrow 0$

→ gledamo ponavljanja sa početka S niza

→ za element na poziciji i gledamo dale se naredni karakter ponavlja sa narednim karakterom sa početka

S: a b a b a c  
i

$S[i] = S[0]$

$S[i+1] = S[1]$

...

J=0;

while( $S[i+J] = S[J]$ )

J++;

Z[i] = J;

→ pošto je  $Z[i] = J$  ne treba nam pomoćna promenljiva J pa dobijamo:

while( $S[i+Z[i]] == S[Z[i]]$ )  
Z[i]++;

→ kako bi poboljšali složenost, pomicemo granice intervala koji ima najdešnju granicu, koji smo prihvatili kao bi ponovo mogli da konstruimo reč izračunate vrednosti

S: a b a b a c  
Z: 0 0 3 0 1

Z: 0 0 3 0 1

→ ovdje ne možemo prepisati 3 jer je a na kraju intervala, ali možemo garantovati da se bar 1 karakter ponavlja

→ pošto je u pitanju isti string Z za index od 6 ne moramo računati, već ga samo možemo prepisati

S: a b a c a b a b a c

Z: 0 0 1 0 3 0 1 0 1 0

→ proširujemo interval, proverimo da li može da se proširi (ne može)

S: a a a c a a a a a c

Z: 0 2 1 0 3 2 2 2 1 0

→ ne može 2 jer bi onda ispal iz intervala  
→ proverimo za karakter 3 van intervala može



## \* Implementacija:

```
vector<int> z-niz (string s) {
```

```
 int n = s.size();
```

```
 vector<int> z(n); → inicijalno sve na 0
```

```
 int l = 0;
```

```
 int d = 0;
```

```
 for (int i = 1; i < n; i++) {
```

```
 if (i <= d) { → ako je i u okviru trenutnog intervala onda
 postavljamo z[i] na već sračunatu vrednost
 z[i] = min(z[i-l], d-i+1); } sa početka ili na vrednost
 koja ne ispada iz intervala
```

```
 while (i+z[i] < n && s[i+z[i]] == s[z[i]])
 z[i]++;
```

```
 if (i+z[i]-1 > d) { → ako smo prošli trenutni interval
 prelazimo u naredni
```

```
 l = i;
```

```
 d = i+z[i]-1; }
```

```
 return z; }
```

→ da bismo pretražili pojavljivanja niza s u nizu t, nadovešćemo ih jedan na drugi novim pomoćnim karakterom i upotrebicemo z niz:

s: a b c

t: a b a b c x a b c

s.size()+1 → početak stringa t koji pretražujemo

p: a b c # a b a b c x a b c

z: 0 0 0 0 2 0 3 0 0 0 3 0 0

prema dužini vidimo da li se s pojavljuje u t

```
main() {
```

```
 string s, t;
```

```
 cin >> s >> t;
```

```
 string p = s + "#" + t;
```

```
 vector<int> z = z-niz(p);
```

```
 for (int i = s.size()+1; i < p.size(); i++) {
```

```
 if (z[i] == s.size()) {
```

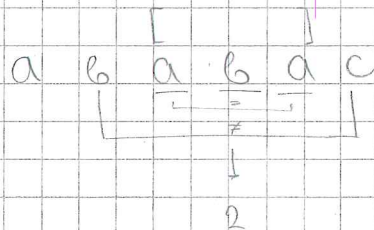
```
 cout << "Pozicija u t: " << i - (s.size()+1); }
```

```
 return 0; }
```

→ ovo će raditi u linearnoj složenosti zbog l i d



## 2° Manacherov algoritam - palindromi:



nećemo gledati samu dužinu palindroma  
već njegov "poluprečnik":

$$a b a \neq 3$$

$$a b a = 2$$

s: a b a b a c

p: 1 2 3 2 1 1

↓  
i

→ p[i]:

$$S[i+1] == S[i-1]$$

$$S[i+2] == S[i-2]$$

ne proveravamo da li je karakter  
jednak samom sebi.

→ j=1

$$\text{while}(S[i+j] == S[i-j])$$

$$j++;$$

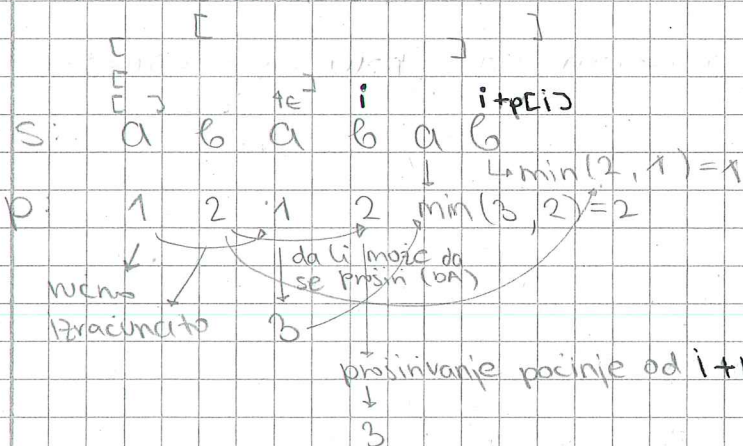
$$p[i] = j;$$

$$\text{while}(S[i+p[i]] == S[i-p[i]])$$

$$p[i]++;$$

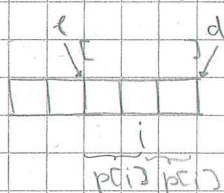
moramo voditi i računa o  
granicama niza pri implementaciji

→ kao i kod 2 niza prat ćemo palindrom koji ima najdešnju granicu  
radi efikasnosti



→ granice azviramo:

$$l = i - p[i] + 1$$

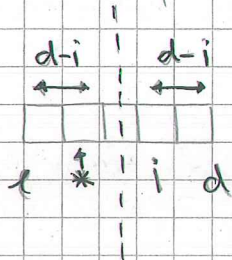


$$l = i - (p[i] - 1)$$

$$d = i + (p[i] - 1)$$

→ ±1 kako bi uhvatili  
prvi i poslednji element  
unutar intervala

→ određivanje simetričnih polja kada i-ti element upada u interval:



indeks p niza

$$* = l + d - i$$

→ pojedinačna udaljenost  
od krajeva intervala

→ ovdje isto moramo paziti na granice, pa za p[i] gledamo da li  
ispada iz intervala sa:

$$p[i] = \min(p[l+d-i], d-i+1);$$



## ⊗ Implementacija:

```
vector<int> manacher(string s) {
```

```
 int n = s.size();
```

```
 vector<int> p(n, 1);
```

```
 int l = 0;
```

```
 int d = 0;
```

svaki karakter je jednak samom sebi pa je silma početna vrednost 1

```
 for(int i = 1; i < n; i++) {
```

```
 if(l <= d) {
```

→ ako trenutni karakter upada u interval uzmemo vrednost njegovog "ogledala"

```
 p[i] = min(p[l+d-i], d-i+1);
```

```
 while(i-p[i] >= 0 && i+p[i] < n &&
```

```
 s[i+p[i]] == s[i-p[i]]) {
```

```
 p[i]++;
```

```
 if(i+p[i]-1 > d) {
```

→ -1 jer smo već unijeli centar sa i?

```
 l = i - (p[i]-1);
```

```
 d = i + (p[i]-1);
```

```
 return p;
```

→ ovaj implementiran algoritam radi **SAMO** za palindrome neparne dužine

S: a b b a

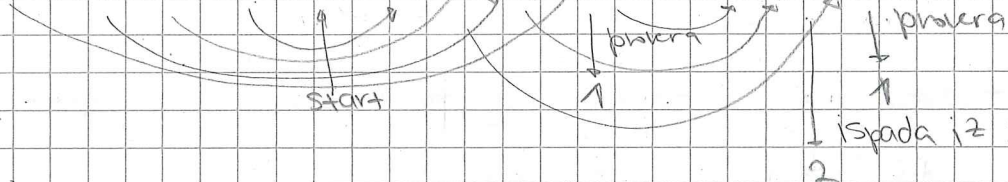
p: 1 1 1 1

→ ovaj problem rešavamo tako što hoćemo da imamo centar unutar palindroma parne dužine, a pošto ne znamo gde se tačno tački palindromi nalaze u nizu, dodaćemo naš pomoćni karakter između svaka dva karaktera čime ćemo moći da pronademo palindrome svih dužina

abba    bab    aba

S: # a # b # b # a # b # a #

p: 1 2 1 2 5 2 1 4 1 4 1 4 1



→ sve što ima  $p[i] > 2$  jeste palindrom

→ dužina palindroma će biti  $2 \cdot p[i] - 1$ , tj. dva puta poluprečnik minus jedan (centralni) karakter jer ga brojimo 2 puta (ao je sa #)

→ dužina palindroma bez # je  $\frac{2 \cdot p[i] - 1}{2}$ , ali pošto je

devenje celobrojno, to će biti  $p[i] - 1$



```
int main() {
 string S;
 cin >> S;
 string t;
```

```
t = "#";
for (char c : S) {
 t += c;
 t += "#";
}
```

```
vector<int> p = manacher(t);
```

//maksimalna duzina palindroma

oduzimamo pocetak da bi dobili iterator

```
int i = max_element(begin(p), end(p)) - begin(p)
cout << p[i] - 1; *
```

→ ako hocemo da ispisemo pronaden podstring (palindrom) potreban nam je indeks pocetnog karaktera, u originalnom stringu

0 1 2 3 4 5 6 7 8 9 10 11 12  
[a b b a] b a → #a#b#b#a#b#a#  
↑ ↑ ↑ ↑  
 $\frac{1}{2}=0$   $\frac{3}{2}=1$   $\frac{5}{2}=2$   $\frac{7}{2}=3$   
p[i]=5

$$\rightarrow i_s = \frac{i_t}{2}$$

i-p[i]+1 i  
#a#b#b#a#  
p[i] p[i]

→ pocetak\_t = centar - palindroma - p[centar - palindroma] + 1;

$$pocetak_s = \frac{pocetak_t}{2}$$

⊗

//ispis najduzeg palindroma:

```
int pocetak_t = i - p[i] + 1;
int pocetak_s = pocetak_t / 2;
cout << s.substr(pocetak_s, p[i] - 1) << "/h";
return 0; }
```

→ umesto [i, d] intervala mozemo raditi i sa [i, R] intervalom gde nam je C indeks centra palindroma, a R indeks desnog kraja

→ **BITNO** - pocetne vrednosti niza su 0, ne 1! ne racunamo samo samog sebe!

→ indeks mirnog elementa:

i-c i-c  
| | | | |  
↓ ↓ ↓  
\* c i

$$c = * + i - c$$

$$2c - i = * \rightarrow mirror = 2c - i$$



# \* Implementacija:

// tražimo ukupan broj palindroma niske S

```
int manacher(string &S){
```

```
 string t;
```

```
 t = "#";
```

```
 for(char c : S){
```

```
 t += c;
```

```
 t += "#"; }
}
```

```
int n = t.size();
```

```
vector<int> p(n); → brojac počinje od 0
```

```
int C=1;
```

```
int R=1;
```

```
int mirror;
```

```
for(int i=1; i<n; i++){
```

```
 mirror = 2C - i;
```

```
 if(i < R) { → samo <, ne <= kao u prethodnom
```

```
 p[i] = min(p[mirror], R - i); } → nema +1
```

```
while(i - (p[i]+1) >= 0 && i + (p[i]+1) < n &&
```

```
 t[i - (p[i]+1)] == t[i + (p[i]+1)]){
```

```
 p[i]++; }
```

```
if(i + p[i] > R) { → nema -1 jer je centar samo
```

```
 jednom unjučen
```

```
 C = i;
```

```
 R = i + p[i]; }
```

```
int sum = 0
```

```
for(int x = p)
```

```
 sum = x/2;
```

```
return sum; }
```

→ ne radimo sum++ za svako x >= 2  
jer želimo da prebrojimo i palindrome  
unutar palindroma;

```
int main(){
```

```
 string s;
```

```
 cin >> s;
```

```
 cout << manacher(s);
```

```
 return 0; }
```

aabaa:

aa

aba

→ ako bi radili samo sum++  
dobili sum=3, a ne 4

aa

#a# se neće računati jer ima  
dužinu 1 jer smo ušli od 0:

#a#

0 1 0 → 1/2 = 0



## II Geometrijski algoritmi:

→ imamo par tačaka  $x, y$

using  $tačka = \text{pair}(\text{int}, \text{int})$ ;

→ vektor  $AB$  dobijamo  $AB = (x_B - x_A, y_B - y_A)$ ;

→ za predstavljanje vektora ćemo koristiti istu strukturu tačka

tačka vektor (tačka  $a$ , tačka  $b$ ) {

return {  $b.\text{first} - a.\text{first}, b.\text{second} - a.\text{second}$  }; }

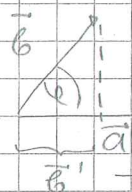
→ norma / dužina vektora  $AB$ :

int norma (tačka  $a$ , tačka  $b$ ) {

tačka  $AB = \text{vektor}(a, b)$ ;

return  $AB.\text{first} * AB.\text{first} + AB.\text{second} * AB.\text{second}$ ; }

→ skalarni proizvod vektora  $\vec{a}$  i  $\vec{b}$ :



$$\vec{a} \cdot \vec{b} = x_a \cdot x_b + y_a \cdot y_b$$

$$\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cdot \cos \varphi$$

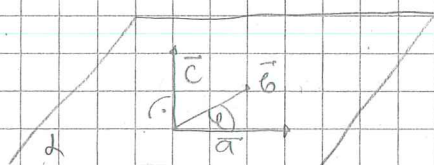
$$\vec{b}' \rightarrow |\vec{b}'| = |\vec{b}| \cdot \cos \varphi \rightarrow \vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}'|$$

→ projekciju vektora  $b$  na vektor  $a$  ( $\vec{b}'$ ) možemo dobiti:

$$|\vec{b}'| = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}|}$$

→ korisno da ne bi računali  $\cos \varphi$

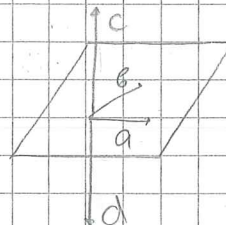
→ vektorski proizvod  $\vec{a}$  i  $\vec{b}$ :



$$\vec{a} \times \vec{b} = \vec{c}, \quad \vec{c} \text{ vektor normalan na ravan } d$$

→ na prvi pogled izgleda besmisleno jer u računanju ne radimo sa 3D prostorom, ali će biti korisno za računanje orijentacije

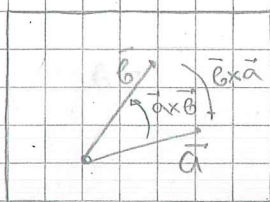
$$|\vec{a} \times \vec{b}| = |\vec{a}| |\vec{b}| \sin \varphi$$



$$c = \vec{a} \times \vec{b}$$

$$d = \vec{b} \times \vec{a}$$

→ vektorski proizvod u 2D:



$\vec{a} \times \vec{b}$  - ide u nas

$\vec{b} \times \vec{a}$  - probada papir

→ računamo vektorski proizvod tako što se "pretvaramo" da je treća dimenzija 0

$$\vec{a} \times \vec{b} = x_a y_b - x_b y_a$$

$> 0$  imamo pozitivno orijentisanje

$< 0$  imamo negativno orijentisanje



→ geometrijski, vektorski proizvod predstavlja površinu paralelograma razapetog vektorima  $\vec{a}$  i  $\vec{b}$



→ ako  $\vec{a} \times \vec{b} = 0$  onda su  $\vec{a}$  i  $\vec{b}$  kolinearni

```
int skalarni(tacka & a, tacka & b) {
 return a.first * b.first + a.second * b.second; }
```

```
int vektorski(tacka & a, tacka & b) { → BITAN redosled
 return a.first * b.second - b.first * a.second; }
```

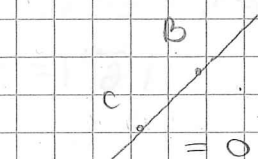
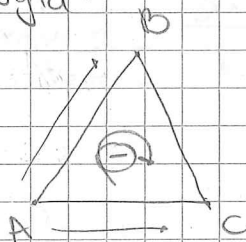
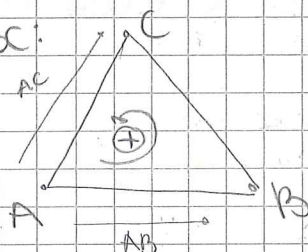
```
int orijentacija(tacka & a, tacka & b, tacka & c) {
 tacka AB = vektor(a, b);
 tacka AC = vektor(a, c);
```

```
 return vektorski(AB, AC); }
 > 0 - (+)
 < 0 - (-)
```

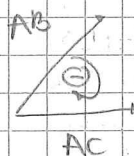
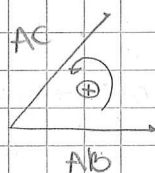
```
bool kolinearne(tacka & a, tacka & b, tacka & c) {
 return orijentacija(a, b, c) == 0; }
```

→ za orijentaciju trougla:

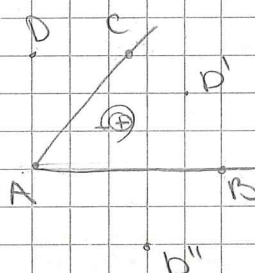
$\triangle ABC$ :



$AB \times AC$



→ možemo ispitati pripadnost tačke trouglu pomoću orijentacije  
→ počnimo od pripadnosti uglu



$ABC \oplus$

$AD'C \oplus$

$AB''C \oplus$

$ABD \oplus$

$ABD' \oplus$

$ABD'' \ominus$

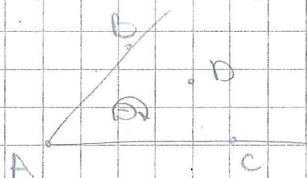
↓  
 $D' \in \angle CAB$

→ uslov za  $D' \in \angle CAB$ : 1°  $O(A, B, D) \geq 0$

2°  $O(A, D, C) \geq 0$



problem: mnogostajna orijentacija ugla



$D \in \angle CAB$ , ali  $ABD \oplus$  ✓

→ resenje: prava orijentacije ugla i zamena tačaka uobliku je orijentacija negativna

```
bool unutarUgla(tačuna a, tačuna b, tačuna c, tačuna d) {
```

```
 if (orijentacija(a, b, c) < 0)
 swap(b, c);
```

```
 return orijentacija(a, b, d) >= 0 && orijentacija(a, d, c) >= 0; }
```

→ tačka je unutar trougla ako je unutar 2 njegova ugla

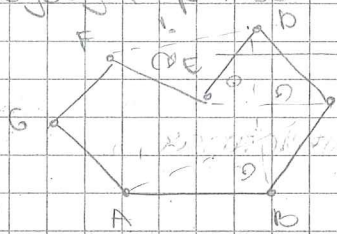
```
bool unutarTrougla(tačuna a, tačuna b, tačuna c, tačuna d) {
```

```
 return unutarUgla(a, b, c, d) && unutarUgla(b, c, a, d); }
```

→ provera konvexnosti mnogougla pomoću orijentacije



Svake 3 uzastopne tačke će biti pozitivne orijentacije



→ negativna orijentacija

→ problem: ako je ABCDE  
zadato kao EDCBA

→ resenje: svake tri tačke moraju imati istu orijentaciju

```
bool ispitajKonvexnost(vector<tačuna> & p) {
```

```
 bool pos = false;
```

```
 bool neg = false;
```

```
 for (int i = 0; i < p.size(); i++) {
```

```
 int j = (i + 1) % p.size();
```

```
 int k = (i + 2) % p.size();
```

```
 int smer = orijentacija(p[i], p[j], p[k]);
```

```
 if (smer < 0)
```

```
 neg = true;
```

```
 if (smer > 0)
```

```
 pos = true; }
```

```
 return !(pos && neg); }
```

→ kada stignemo do kraja vektora  
kao da ne bi ispalili iz njega, već  
nastavili sa početka



→ pravljenje prostog mnogougla od zadatih tačaka

- želimo redosled tačaka takav da se duži mnogougla međusobno ne seku
- izabraćemo jednu tačku iz našeg skupa u odnosu na koju ćemo sortirati sve ostale tačke
- izabrana tačka će biti najlevija tačka, a ako postoji više tačaka sa najmanjom X koordinatom, biramo onu sa najvećom Y koordinatom
- za samo sortiranje tačaka koristićemo orijentaciju trougla, tj. želimo da sve orijentacije budu pozitivne
- ako su dve tačke kolinearne, vraćamo onu koja je bliža izabranoj tački

```
void prostMnogougao(vector<tačka> & t) {
```

```
 auto max = max_element(begin(t), end(t),
```

```
 [](const tačka & a, const tačka & b) {
```

```
 return a.first < b.first ||
```

```
 (a.first == b.first && a.second > b.second);
```

```
 }); swap(*begin(t), *max); → postavljamo izabranu tačku
 tačka to = t[0]; za prvu tačku mnogougla
```

```
 sort(begin(t)+1, end(t), [to](tačka & a, tačka & b) {
```

```
 auto smer = orijentacija(to, a, b);
```

```
 if (smer > 0)
```

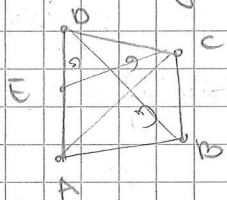
```
 return true;
```

```
 else if (smer < 0)
```

```
 return false;
```

```
 return normal(to, a) <= normal(to, b);
```

- ako na kraju postoji grupacija tačaka kolinearnih sa to, moramo da ih obrnemo redosled kako bi pravilno zatvorili mnogougao



$smer(ADE) = 0$

$AE < AD \rightarrow ABCE$

↓

$smer(ADE) = 0$

$reverse(DE) \rightarrow ABCDE$

⊗

```
 auto it = end(t) - 1;
```

```
 while (orijentacija(*(it-1), *it, to) == 0)
```

```
 it = it - 1;
```

```
 reverse(it, end(t));
```



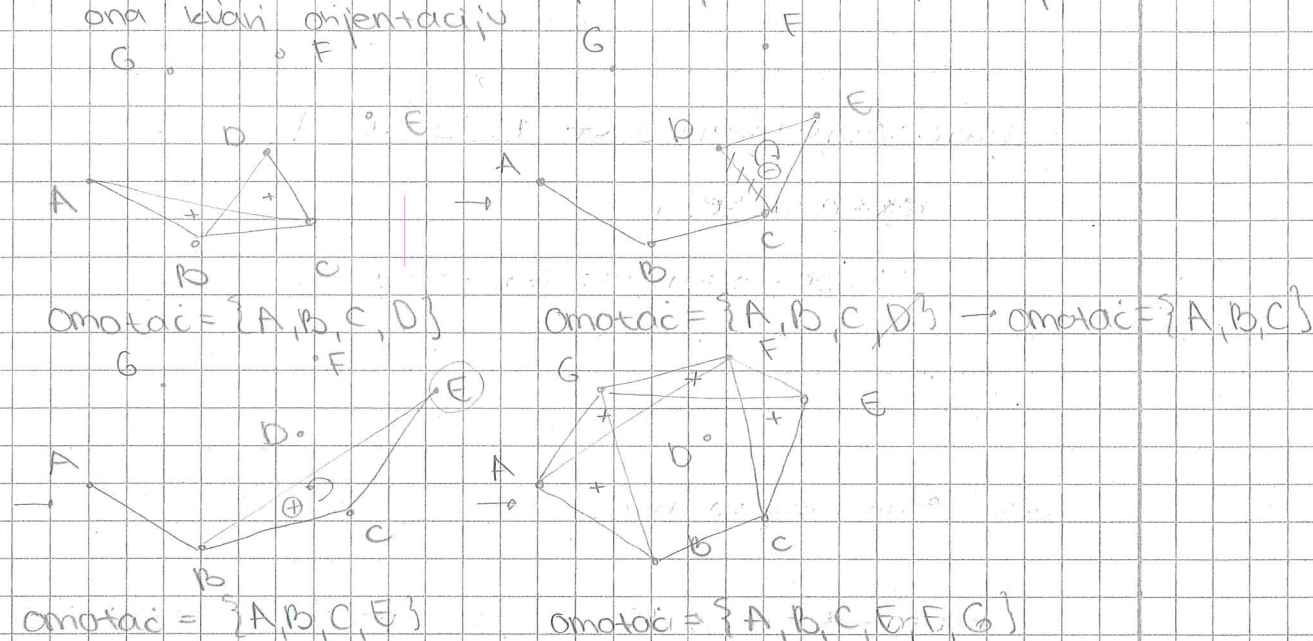
→ pravljenje konveksnog omotaca oko grupe tačaka

→ za rešavanje ovog problema postoji algoritam Graham Scan koji radi u  $O(n \log n)$  složenosti

→ ideja: biramo jednu ekstremnu tačku i sortiramo ostale u odnosu na nju (kao u prethodnom zadatku)

→ izgradnju omotaca onda vršimo nad sortiranim tačkama tako što u omotac dodajemo tačke dokle god slave utastopne 3 prave konveksan luk, tj. imaju pozitivnu orijentaciju

→ kada stignemo do tačke čija je orijentacija negativna, izbacujemo iz omotaca poslednje dodatu tačku jer ona kvari orijentaciju



→ ovakvo uklanjanje tačaka iz omotaca će se vršiti dokle god je orijentacija negativna

```
vector<int> grahamScan(vector<tačka> & p) {
```

```
 swap(p[0], *min_element(begin(p), end(p)));
 sort(begin(p)+1, end(p),
 [&](tačka a, tačka b) {
```

nalazi tačku sa min X koordinatom ili u slučaju više takvih, sa min Y koordinatom

```
 return orijentacija(p[0], a, b); } });
```

```
 vector<int> omotac = {0};
 for(int i=1; i<p.size(); i++) {
```

//uklanjanje poslednje tačke dok je orijentacija negativna

```
 while(omotac.size() > 1 &&
 orijentacija(p[omotac[omotac.size()-2]], p[omotac.back()], p[i]) < 0) {
 omotac.pop_back(); }
```

```
 omotac.push_back(i); }
```

```
 return omotac; }
```



- da li je tačka unutar konvexnog mnogougla možemo proveriti tako što proverimo da li je unutar nekog od trouglova koje grade tačke mnogougla
- ovo je najefikasnije rešiti binarnom pretragom jer svaka dijagonala mnogougla ga deli na dva manja, a ako tačka nije unutar jednog od njih, celu tu polovinu možemo odbaciti

```
bool unutarMnogougla(vector<tačka> &CH, tačka &S) {
 int n = CH.size();
```

```
//provera da li je S van prostora mnogougla određenog
//tačkama 0, 1 i n-1:
```

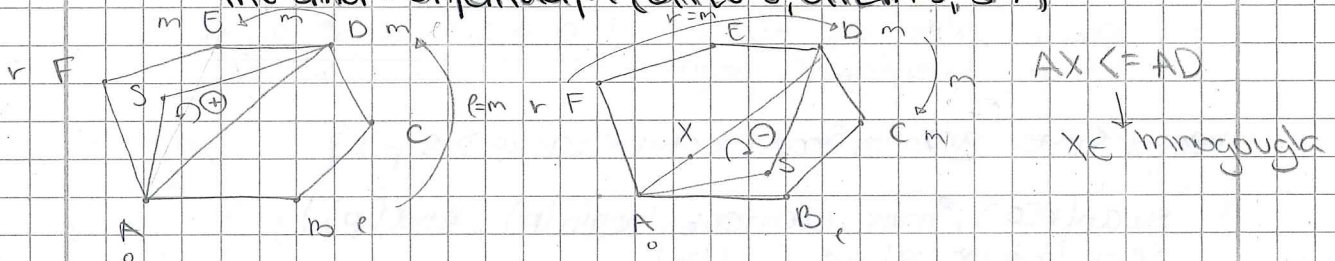
```
if (orientacija(CH[0], CH[1], S) < 0 ||
 orientacija(CH[0], CH[n-1], S) > 0)
 return false;
```

```
//sada za l i r možemo koristiti tačke 1 i n-1
```

```
int l = 1;
int r = n - 1;
```

```
//radimo ovo dok ne sužimo prostor toliko da sadrži
//samo jedan trougao polaznog mnogougla
```

```
while (r - l > 1) {
 int m = l + (r - l) / 2; // prepolovimo trenutni prostor
 int smer = orientacija(CH[l], CH[m], S);
```



```
if (smer > 0)
 l = m;
else if (smer < 0)
 r = m;
else {
```

→ Situacija kada je tačka baš na pravi kojim smo prepolovili prostor, tada gledamo rastojanje

```
if (norma(CH[0], S) <= norma(CH[0], CH[m]))
 return true;
```

```
else
 return false; }
```

```
//provera da li je tačka unutar trougla do kojeg smo došli:
```

```
if (unutarTrougla(CH[l], CH[r], CH[r], S))
 return true;
```

```
return false; }
```

dobij ray casting!



### III Algebarski algoritmi:

#### 1. Kongruencija po modulu:

$$5 + 10 = 15$$

$$5 + 10 = 3$$

$$5 \equiv_4 1$$

$$10 \equiv_4 2$$

$$\begin{array}{cccccccccccccccc} -7 & -6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

$$\text{mod } 4: \begin{array}{cccccccccccccccc} 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \end{array}$$

$$\rightarrow -3 + 4 = 1$$

$$-5 + 4 = -1 \rightarrow -1 + 4 = 3$$

$$\otimes X \equiv y \pmod{m} \quad \in [0, m)$$

$$X - y = k \cdot m, \quad m \in \mathbb{Z}$$

#### $\otimes$ implementacija sabiranja i oduzimanja po modulu m:

- ako u C++ radimo mod nekog negativnog broja bez obezbeđivanja da pređemo u pozitivne, dobićemo netačno rešenje, npr:

$$-5 \% 4 \rightarrow -1 \text{ što nije tačno}$$

- u C++ se dešava sledeće:

$$-5 \% 4 \rightarrow 5 \% 4 = 1 \rightarrow 1 \cdot (-1) = -1$$

- ovo izbegavamo obezbeđivanjem da naš broj sigurno bude unutar intervala od 0 do m
- zbog toga pravimo pomoćnu funkciju:

```
int mod(int a, int m) {
 if (a >= 0)
 return a;
 else
 return (m + a % m) % m;
}
```

- $a \% m$  sigurno vraća broj iz intervala  $[0, m)$ , ali C++ dodaje  $(-1)$  ako je broj negativan, pa to poništavamo dodavanjem još jednog  $m$
- dodatno  $\% m$  radimo zbog slučaja kada je  $a = -m$ , da nemamo  $\% m$  na kraju onda bi dobili  $m + (-m) \% m = m + 0 = m$  što nije korektno
- sabiranje po modulu m sada izgleda:

```
int add(int a, int b, int m) {
 return mod(a + b, m);
}
```



## ⊗ implementacija množenja po modulu m:

- ovde moramo paziti na pretvaranje pri množenju a i b, pa prvo želimo same a i b da sledimo na njihov modul od m:

```
int mul(int a, int b, int m){
 return mod(mod(a, m) * mod(b, m), m); }
```

## ⊗ primer upotrebe - linearni kongruentni generator pseudo-slučajnih brojeva

- ideja - zadamo početni broj, a svaki naredni računamo po nekoj formuli

$$X_0 = 1$$
$$X_1 = (a \cdot X_0 + c) \bmod m$$

→ a, c i m su neki parametri koji se biraju unapred, npr.  $a=10$ ,  $c=6$ ,  $m=19$

$$X_0 = 1$$
$$X_1 = (10 \cdot 1 + 6) \bmod 19 = 16$$
$$X_2 = (10 \cdot 16 + 6) \bmod 19 =$$
$$\dots$$

$X_0$  - zadati parametar od kog zavisi ponavljanje niza  
 $X_0$  - seed

## → implementacija

```
int main(){
 int a=10, c=6, m=19;
 int seed=1;

 int x=seed;
 do{
 x=add(mul(a, x, m), c, m);
 cout << x << " ";
 }while(x != 1);
 return 0; }
```

$x = (a \cdot x + c) \% m$

## 2. Pronalazak najvećeg zajedničkog delioca:

- ako imamo razlomak  $\frac{a}{b}$  i želimo da ga svedemo na gde su  $a_1$  i  $b_1$  međusobno prosti, moramo naći njihov najveći zajednički delilac, npr:

$$a=60 \quad b=16 \quad \frac{60}{16} = \frac{15}{4} \rightarrow \text{nzd}(60, 16) = 4$$

## ⊗ Euklidov algoritam:

$d \mid a, b \rightarrow d$  mora da deli a }  $d$  mora da deli i  $a-b$ :  
 $\rightarrow d$  mora da deli b }

$$a = a' \cdot d \rightarrow a - b = (a' - b') \cdot d$$
$$b = b' \cdot d$$

- iz prethodnog  $d \mid a - 2b$ :  
 $a - 2b = (a' - 2b') \cdot d \Rightarrow d \mid a - k \cdot b$



- Za naš konkretni primer:

$$\begin{array}{l} a=60 \\ b=16 \end{array}, \begin{array}{l} d \mid 60 \\ d \mid 16 \end{array} \quad \begin{array}{l} d \mid 60 - 16 = 44 \\ d \mid 60 - 2 \cdot 16 = 28 \\ d \mid 60 - 3 \cdot 16 = 12 \end{array} \quad \left. \vphantom{\begin{array}{l} a=60 \\ b=16 \end{array}} \right\} 60 \bmod 16 = 12$$

$$60 - 3 \cdot 16 = 12$$

ostatak  
pri deljenju  
60 sa 16

→ dobili smo broj manji od 16, ali pošto  
vozi da  $d$  deli i 16 i 12 možemo  
da nastavimo proces sa ova dva broja.

$$\begin{array}{l} d \mid 16 - 12 = 4 \\ d \mid 12 - 4 = 8 \\ d \mid 12 - 2 \cdot 4 = 4 \\ d \mid 12 - 3 \cdot 4 = 0 \end{array} \quad \left. \vphantom{\begin{array}{l} d \mid 16 - 12 = 4 \\ d \mid 12 - 4 = 8 \end{array}} \right\} \begin{array}{l} 16 \bmod 12 = 4 \\ 12 \bmod 4 = 0 \end{array}$$

→ dobijamo sekvencu:

$$a, b, a \div b, b \div (a \div b), \dots, 0$$

→ u našem primeru: 60, 16, 12, 4, 0

biramo 2a  $\text{NzD}$

- implementacija:

```
#include <tuple>
```

```
int Euklid(int a, int b) {
```

```
 while (b > 0) {
```

```
 tie(a, b) = make_pair(b, a % b);
```

```
 return a;
```

### \* Proširenje Euklidovog algoritma:

- $\text{NzD}$   $d$  se uvek može napisati kao:

$$d = \text{nzd}(a, b)$$

$$4 = \text{nzd}(60, 16)$$

$$d = a \cdot x + b \cdot y, \quad x, y \in \mathbb{Z}$$

$$4 = x \cdot 60 + y \cdot 16$$

→ npr:  $4 = 60 \cdot (-1) + 16 \cdot 4$

$$\begin{array}{l} d \mid a \Rightarrow d \mid a \cdot x \\ d \mid b \Rightarrow d \mid b \cdot y \end{array} \quad \left. \vphantom{\begin{array}{l} d \mid a \Rightarrow d \mid a \cdot x \\ d \mid b \Rightarrow d \mid b \cdot y \end{array}} \right\} d \mid \underline{ax + by}$$

ovaj deo MOGA  
biti deljiv sa  $d$

→ Zanima nas koje sve vrednosti  $x$  i  $y$  mogu uzimati

$$\begin{array}{l} a=60 \\ b=16 \end{array}$$

$$\begin{array}{l} 60 = 3 \cdot 16 + 12 \\ 16 = 1 \cdot 12 + 4 \\ 12 = 3 \cdot 4 + 0 \end{array}$$

→ hajemo od ovoga da napravimo  
tablicu iz koje možemo čitati  
 $x$  i  $y$ :



|                        | x | y | želimo da važi: |
|------------------------|---|---|-----------------|
| $a = 60$               | 1 | 0 | $ax + by = a$   |
| $b = 16$               | 0 | 1 | $ax + by = b$   |
| $60 = 3 \cdot 16 + 12$ |   |   | $ax + by = 12$  |
| $16 = 1 \cdot 12 + 4$  |   |   | $ax + by = d$   |

- Za prve dve kolone je bilo izračunati dok ćemo ostale vrednosti dobiti induktivnim algoritmom

$$r = ?$$

$$\text{dato } \left\{ \begin{array}{l|l|l|l} r_1 = a & x_1 = 1 & y_1 = 0 & ax_1 + by_1 = r_1 \\ r_2 = b & x_2 = 0 & y_2 = 1 & ax_2 + by_2 = r_2 \\ r_1 = qr_2 + r & x & y & ax + by = r \end{array} \right.$$

$$\downarrow r = r_1 - qr_2 \rightarrow \text{zamenimo } r_1 \text{ i } r_2 \text{ jednačinama}$$

$$r = (ax_1 + by_1) - q(ax_2 + by_2)$$

$$r = a(\underbrace{x_1 - qx_2}_x) + b(\underbrace{y_1 - qy_2}_y)$$

$$\otimes r = ax + by \quad \text{za} \quad \begin{array}{l} x = x_1 - qx_2 \\ y = y_1 - qy_2 \end{array}$$

- implementacija:

int proširen-Euklid (int a, int b, int\* x, int\* y) {

int r1 = a, r2 = b;  
int x1 = 1, y1 = 0;  
int x2 = 0, y2 = 1;

while (r2 > 0) {  
int q = r1 / r2;  
tie(r1, r2) = make\_pair(r2, r1 - q\*r2);  
tie(x1, x2) = make\_pair(x2, x1 - q\*x2);  
tie(y1, y2) = make\_pair(y2, y1 - q\*y2);  
}

x = x1;  
y = y1;  
return r1; }

### $\otimes$ Linearne Diofantove jednačine:

- one su oblika  $ax + by = c$
- želimo da rešimo ovu jednačinu, a za to imamo 2 slučaja

1°  $d \nmid c$ ,  $d = \text{nzd}(a, b)$   
 $\rightarrow$  j-na nema rešenja

2°  $d \mid c$ ,  $d = \text{nzd}(a, b)$   
 $\rightarrow$  j-na ima beskonačno rešenja



- ako rešenje postoji onda ga nalazimo na naredni način:

$$ax' + by' = d \quad / \quad \frac{c}{d} \quad , \quad \frac{c}{d} - \text{ceo broj jer } d | c$$

$$a \left( x' \cdot \frac{c}{d} \right) + b \left( y' \cdot \frac{c}{d} \right) = c$$

$$ax + by = c$$

- ako su  $a < 0$  ili  $b < 0$  onda:

$$\begin{aligned} (-a)x + (-b)y &= d & \{ \text{a i b moraju} \\ a(-x) + b(-y) &= d & \text{biti } > 0 \text{ jer} \\ & & \text{Euklidov} \\ & & \text{podatak} \end{aligned}$$

→ izračunamo  $x'$  i  $y'$  pomoću proširenog Euklidovog algoritma, i samo to rešenje pomnožimo sa  $c/d$ .

- implementacija:

false - nema rešenja  
true - ima rešenja

bool diofant(int a, int b, int c, int& x, int& y) {

int d = proširen\_Euklid(abs(a), abs(b), x, y);

if (c % d != 0)  
return false;

if (a < 0)  
x \*= -1;

if (b < 0)  
y \*= -1;

ako su a ili b negativni  
menjamo znak x ili y

x \*= c/d;  
y \*= c/d;  
return true; }

### \* NZS pomoću Euklidovog algoritma:

int nzs(int a, int b) {

return (a / Euklid(a, b)) \* b; }

### \* Kongruencija po modulu - deljenje:

$$\frac{b}{a} = x \rightarrow a \cdot x = b$$

$$a \cdot x \equiv b \pmod{m}$$

$$a \cdot x - b = y' \cdot m, \quad y' \in \mathbb{Z}$$

$$a \cdot x - y' \cdot m = b$$

$$-y' = y \rightarrow ax + ym = b$$

ovde samo tvrdimo da postoji neko  $y'$  za koje jednakost važi pa možemo uvesti drugu oznaku za  $y'$

→ ovde rešenje tj. vrednost x dobijamo rešavanjem Diofantove jednačine



## ⊗ primer primene - pokvaren sat:

- za 1h stvarnog vremena sat se pomeri za  $a^h$ , npr.:

 - stvarno vreme

 - sat vreme prema  $a^h$

$$a = 7$$

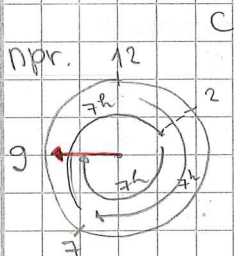


→ zadatak - procitati koliko je zaista sati

$C$  - koliko trenutno piše da je sati

$t$  - koliko je zaista vremena proteklo

$t \rightarrow a \cdot t$  (za  $t^h$  sat se pomerio za  $a \cdot t$ )



$$a = 7$$

$$7 \cdot t \equiv 9 \pmod{12}$$

$$\rightarrow t = 3 \text{ jer } 7 \cdot 3 = 21 \pmod{12} = 9$$

jer sat ima 12h

- u opštim brojevima:

$$a \cdot t \equiv C \pmod{12}$$

$$a \cdot t + 12 \cdot y = C \rightarrow \text{linearna Diofantova jednačina}$$

```
int main() {
 int a, c;
 int m = 12;
 cin >> a >> c;
```

```
 int t, y;
```

```
 if (diofant(a, m, c, t, y))
```

```
 cout << "Sada je " << mod(t, 12) << " sati.\n";
```

```
 else
```

```
 cout << "Sat je baš pokvaren.\n";
```

```
 return 0; }
```

- kada nema rešenja:

$$3 \cdot t + 12y = 6 \rightarrow \text{NZD}(3, 12) = 3 \rightarrow 3 \mid 6$$

$$3 \cdot t + 12y = 8 \rightarrow \text{NZD}(3, 12) = 3 \rightarrow 3 \nmid 8 \rightarrow \text{nema rešenja}$$

- za  $a \cdot t + 12y = C$  ćemo uvek imati rešenje ako su  $a$  i  $12$  uzajamno prosti ( $\text{NZD}(a, 12) = 1$ ), a 1 deli svako  $C$ , tj. za npr primer  $a$  može biti  $\{1, 5, 7, 11\}$ .



### 3. Eratostenovo Sito:

- vraća bool vektor koji je true za sve proste brojeve manje od n

```
vector<bool> EratostenovoSito (int n){
```

```
 vector<bool> is_prime(n+1, true);
```

```
 is_prime[0] = false;
```

```
 is_prime[1] = false;
```

```
 for (int i=2 ; i*i <= n; i++){
```

```
 if (is_prime[i])
```

```
 for (int j=i*i ; j <= n; j+=i)
```

```
 is_prime[j] = false;
```

```
 return is_prime; }
```